DELAY COMPOSITION THEORY: A REDUCTION-BASED SCHEDULABILITY THEORY FOR DISTRIBUTED REAL-TIME SYSTEMS

BY

PRAVEEN JAYACHANDRAN

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2010

Urbana, Illinois

Doctoral Committee:

Associate Professor Tarek Abdelzaher, Chair
Professor Sanjoy Baruah
Professor P.R. Kumar
Professor Lui Sha

Abstract

This thesis develops a new reduction-based analysis methodology for studying the worst-case end-to-end delay and schedulability of real-time jobs in distributed systems. The main result is a simple delay composition rule, that computes a worst-case bound on the end-to-end delay of a job, given the computation times of all other jobs that execute concurrently with it in the system. This delay composition rule is first derived for pipelined distributed systems, where all the jobs execute on the same sequence of resources before leaving the system. We then derive the delay composition rule for systems where the union of task paths forms a Directed Acyclic Graph (DAG), and subsequently generalize the result to non-acyclic task graphs as well, under both preemptive and non-preemptive scheduling. The result makes no assumptions on periodicity and is valid for periodic and aperiodic jobs. It applies to fixed and dynamic priority scheduling, as long as all jobs have the same relative priority on all stages on which they execute. The delay composition result enables a simple reduction of the distributed system to an equivalent hypothetical uniprocessor that can be analyzed using traditional uniprocessor schedulability analysis to infer the schedulability of the distributed system. Thus, the wealth of uniprocessor analysis techniques can now be used to analyze distributed task systems. Such a reduction significantly reduces the complexity of analysis and ensures that the analysis does not become exceedingly pessimistic with system scale, unlike existing analysis techniques for distributed systems such as holistic analysis and network calculus. Evaluation using simulations suggest that the new reduction-based analysis is able to significantly outperform existing analysis techniques, and the improvement is more pronounced for larger systems.

We develop an algebra, called delay composition algebra, based on the delay composition results for systematic transformation of distributed real-time task systems into single-resource task systems such that schedulability properties of the original system are preserved. The operands of the algebra represent workloads on composed subsystems, and the operators define ways in which subsystems can be composed together. By repeatedly applying the operators on the operands representing resource stages, any distributed system can be systematically reduced to an equivalent uniprocessor that can be analyzed later to determine end-to-end delay and schedulability properties of all jobs in the original distributed system.

The above reduction-based schedulability analysis techniques suffer from pessimism that results from mismatches between uniprocessor analysis assumptions and characteristics of workloads reduced from distributed systems, especially for the case of periodic tasks. To address the problem, we introduce *flow-based mode changes*, a uniprocessor load model tuned to the novel constraints of workloads reduced from distributed system tasks. In this model, transition of a job from one resource to another in the distributed system, is modeled as mode changes on the uniprocessor. We present a new iterative solution to compute the worst-case end-to-end delay of a job in the new uniprocessor task model. Our simulation studies suggest that the resulting schedulability analysis is able to admit over 25% more utilization than other existing techniques, while still guaranteeing that all end-to-end deadlines of tasks are met.

As systems are becoming increasingly distributed, it becomes important to understand their *structural robustness* with respect to timing uncertainty. Structural robustness, a concept that arises by virtue of multi-stage execution, refers to the robustness of end-to-end timing behavior of an execution graph towards unexpected timing violations in individual execution stages. A robust topology is one where such violations minimally affect end-to-end execution delay. We show that the manner in which resources are allocated to execution stages can affect the robustness. Algorithms are presented for resource allocation that improves the robustness of execution graphs. Evaluation shows that such algorithms are able to reduce deadline misses due to unpredictable timing violations by 40-60%. Hence, the approach is important for soft real-time systems, systems where timing uncertainty exists, or where worst-case timing is not entirely verified.

We finally show two contexts in which the above theory can be applied to the domain of wireless networks. First, we developed a bandwidth allocation scheme for elastic real-time flows in multi-hop wireless networks. The problem is cast as one of utility maximization, where each flow has a utility that is a concave function of its flow rate, subject to delay constraints. The delay constraints are obtained from our end-to-end delay bounds and adapted to only use localized information available within the neighborhood of each node. A constrained network utility maximization problem is formulated and solved, the solution to which results in a distributed algorithm that each node can independently execute to maximize global utility. Second, we study the problem of minimizing the worst-case end-to-end delay of packets of flows in a wireless network under arbitrary schedulability constraints. Using a coordinated earliest-deadline-first strategy, we show that a worst-case end-to-end delay bound that has the same form as our delay composition results for distributed systems can be obtained.

We discuss several avenues for future work that build on top of the theory developed in this thesis. We hope that this thesis will provide the foundation to develop a more comprehensive and widely applicable theory for the study of delay, schedulability, and other end-to-end properties in distributed systems.

To my mother.

Acknowledgments

I consider myself extremely fortunate to have had as fine a mentor as Prof. Tarek Abdelzaher. I would like to think I have learned a lot from him over the last five years, not just in matters pertaining to conducting research, but also in several other avenues in which he excels such as teaching, communicating ideas effectively, technical writing, time management, and maintaining cordial relations with everyone he interacts with. I cannot thank him enough for his infinite patience and constant encouragement, especially during certain difficult times. My very first paper that forms the basis for this thesis was rejected twice, and his faith in my abilities and the patience he showed towards my poor writing skills were vital in the paper eventually being accepted and even winning the best student paper award. I will forever cherish the long conversations we have had - some were boisterous arguments over complicated mathematical proofs, and others were soulful discussions on research philosophy and writing style. It is indeed a rude awakening that I'll no longer have him advising me on matters of importance. The excellent rapport we have enjoyed have undoubtedly made this an extremely memorable journey and a very difficult parting.

I have also had the good fortune of having as illustrious names as Prof. Lui Sha, Prof. P.R. Kumar, and Prof. Sanjoy Baruah on my committee. Their ideas and comments have not only helped me improve the work presented in this thesis, but have also sculpted a better researcher out of me. Each one of them have been tremendous role models, and have left me with the ambition of being in their position twenty years from now.

This is a fine opportunity to reflect and thank my mother, Karthiyayini, and my late father, Jayachandran, for their unfailing love and support. They've given me what is perhaps the greatest gift of all - the confidence to undertake and accomplish anything I dare to dream of. They have rejoiced with me on every one of my successes. They have supported and cared for me in all my little failures and have helped me learn from them. I am the person whom they've made me to be.

It was on my brother Prakash's advice that I joined the University of Illinois, for he had spent two years here obtaining a master's degree. I had never imagined at the time, that I would have such a wonderful experience here. I have always followed on his footsteps. We have attended the same primary and high

school, Vidya Mandir, both of us got computer science degrees from the Indian Institute of Technology Madras, India. Now, both of us have degrees from the University of Illinois. Such has his impression on me been, that I've chosen longer programs and spent more time at each place.

I must also thank my cousin Venkat and his wife Deepti who showed great empathy, support and guidance through some very emotionally testing times. I have the greatest respect and love for both of them and eagerly look forward to show them that at every given opportunity.

Life in Champaign would have been drab without the company of my many good friends. We have spent a lot of time together meeting up for every meal, chatting, playing games, pulling each others legs, cooking exorbitant dinners and watching movies. I have shared apartments with Pradeep for all the time I've been here, and his incessant jabber would ensure that the mind is never given a half chance to slip into a state of melancholy. I have enjoyed my time training to run marathons with Aravind, Vikhram, Swetha, Avanija, Preeti, Pradeep, Harini and Vidisha. I was fortunate to find Aravind and Gowri to be as interested in music as I was and we have spent a lot of time listening and discussing music. Vinay, Raghu, and Pradeep have been responsible for me to take a greater interest in cooking, which I'll cherish for a lifetime, especially with me being someone who loves food. I have always yearned for the angelic touch with which Vidisha and Harini prepare chai, something I'll sorely miss when I leave Champaign. Milu and Avanija have brought many a smile on my face by constantly making mouth-watering desserts. I tried my hand at learning swing dancing and enjoyed it with the company of Swetha, Vikhram, and Milu. Chandu, Preeti, Navaneethan and I have spent long nights watching episodes from the gripping television series, the Wire, which I thoroughly enjoyed. Chandu and Preeti's one year old adorable son, Aari, has been the source of entertainment for many an evening. Kaushik, Vikhram, Aravind and I have had many serious discussions and debates on a variety of topics. Although, I have never openly admitted it, I admire Vivek's and Navaneethan's sense of humor and quirkiness. I am thankful to Milu for having introduced me to the habit of going to church every Sunday, and I have come to sincerely appreciate that one hour of peaceful and solemn music. Its only now that I realize how much of an impression a few years of togetherness can have on you. I cannot find words to adequately acknowledge the role my friends have played in making these the best years of my life.

Playing bridge has been a passion for me for several years now. When I came to Champaign five years ago, I knew of a bridge club, but was hoping for nothing more than an occasional game. I remember the first couple of times I played at the club. It was so heartening to see everyone be so friendly and encouraging. I really appreciate how some of the more experienced players took the time to point out my mistakes, which has helped me to significantly improve my game. I must thank my bridge partners Dan Bunde, Bill Lindemann, Paul Holmes, and Terry Goodykoontz for fun-filled bridge sessions every Monday night and

for dragging me along to several out of town tournaments. I must also express my appreciation for all the excellent home-made food that the club members would bring from time to time. Never did I imagine that acquaintances from the bridge table would blossom into real friendships. Bill Lindemann and his family have been kind enough to invite me to their thanksgiving and Christmas family dinners. Bill and I have gone out for dinners and movies and I have thoroughly enjoyed his company. My association with the bridge club gave me a wonderful past time and I can't thank the club and its players enough for making my stay here so wonderful and memorable.

Champaign might be in the middle of hundreds of miles of nothing but corn fields. But, there is something charming and divine about this place and its people that I have come to love so much. The familiarity of the streets and restaurants, the calmness that comes with being in a small town, the chirping of birds in the morning, the cacophony of insects after dusk, the warmth of spring, the blistering summer, the dainty fall colors, and the frigid white winters, all have some role in making me consider this place home.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

We propose a new reduction-based theory called delay composition theory for the analysis of delay and schedulability of real-time jobs in distributed systems. The theory enables the systematic reduction of a distributed system workload to an equivalent hypothetical uniprocessor workload, such that well known uniprocessor schedulability analysis techniques can be used to analyze distributed system schedulability. This reduction-based methodology significantly reduces the complexity of the analysis and tends to be much less pessimistic than existing analysis techniques for large distributed systems.

Real-time applications are becoming increasingly more complex with respect to system scale and the number of resources involved. With Moore's law approaching saturation, and power and reliability issues hampering the growth of multiprocessor systems, the emphasis is shifting towards distributed computation. Avionics and ship-board computing clusters are heading towards increased automation, with several stages of processing for various real-time tasks within a distributed computing environment. Automotive systems have dozens of embedded processors, and tasks such as cruise control and traction control involve several stages of distributed processing, subject to strict timing constraints. Each search query answered by Google, typically goes through thirty different stages of computation, with the server farm comprising of thousands of processors. Manufacturing plants in every industry have several specialized servers, producing hundreds of parts that follow different routes through the system. Cyber-physical systems, as an umbrella term for various personal and military applications, have gained a lot of momentum, with the NSF identifying it as a key focus area for research. In a more abstract setting outside the realm of computing, multi-personnel projects in any industry are also distributed real-time systems. A project typically consists of several sequential tasks with dependencies and precedence constraints amongst them. Each sub-job of a sequential task may be processed by one or a group of people. Here, skilled personnel act as the resources, and the goal of the system is to complete all the tasks within the deadline for the project. Any delay in completing the project directly translates into lost sales, cost overruns, and compromises in product quality. It is thus paramount to have a concrete theoretical understanding of the timing behavior of distributed real-time systems.

Rigorous theory exists today for uniprocessor and multiprocessor systems, while only heuristics are used

to analyze and design larger arbitrary-topology distributed systems. These heuristics are based on intuitions and ideas developed from studying uniprocessor and multiprocessor systems, which tend to be insufficient and on occasion even misinformed in the context of distributed systems. The goal of this thesis is to develop a fundamental understanding of delay and the factors that affect it in distributed real-time systems. In particular, we are interested in determining the worst-case end-to-end delay performance of such systems, as this directly reflects on the extent of resource provisioning and cost involved in maintenance. We hope that the ideas, intuition, and analysis methodology developed through this work will foster further research towards developing a comprehensive theory, and aid in designing more efficient and robust performance-sensitive real-time systems.

In uniprocessor and multiprocessor systems, the natural way to improve system throughput or efficiency, is to make each processor faster (more efficient) and reduce the processing time for each job. Improving the efficiency of a single processor in a multiprocessor system translates into overall improved system throughput and reduced delay. This, however, is not the case in distributed systems. Improving the efficiency of a single resource in the distributed system may not translate into an improvement in the overall system throughput or delay. Consider for instance, a system consisting of five resources and a set of jobs that execute sequentially on each of the five resources. Further, suppose that the third resource is the slowest, and jobs queue up at that resource (there is minimal or no queuing at the other resources). Now, doubling the efficiency of the first resource yields no appreciable improvement in the system throughput as the *bottleneck* resource is still just as slow, and jobs will only have to wait longer before they get serviced at the third resource (in fact, this increases any queuing/inventory costs). However, doubling the efficiency of the bottleneck resource results in a significant reduction in the end-to-end delay. Therefore, identifying the bottleneck becomes extremely important. This, however, is a very challenging problem for large systems with tasks following different routes through the system, as the bottleneck can be different for each task. As we shall show more formally in this thesis, the end-to-end delay of a task is largely a function of the worst-case delay on a single 'hypothetical bottleneck' resource, rather than the cumulative worst-case delay of each resource on which it executes.

Existing techniques for analyzing delay and schedulability of jobs in distributed systems can be broadly classified into two categories: (i) decomposition-based, and (ii) extension-based. The decomposition-based techniques break the system into multiple subsystems, analyze each subsystem independently using current uniprocessor analysis techniques, then combine the results. The extension-based techniques explore ways to extend current uniprocessor analyses to accommodate distributed tasks and resources. Both analysis techniques tend to become increasingly complex and pessimistic with system scale. In contrast, we propose to

use a third category of techniques for analyzing distributed systems that are based on *reduction* (as opposed to decomposition or extension). Rather than breaking up the problem into sub-problems, or extending uniprocessor analyses to more complex systems, we systematically *reduce* the distributed schedulability problem to a single simple problem on a uniprocessor.

A reduction-based approach to system analysis has been devised in many contexts outside distributed system scheduling. For instance, in control theory, there are rules to reduce complex block diagrams into a single equivalent block, which can later be analyzed for stability and performance properties. In circuit theory, laws such as Kirchoff's laws enable complex circuits to be reduced to a single equivalent source and impedance. Apart from reducing the complexity of the problem to that of a single component, such reduction rules also provide fundamental insights into how key performance properties are affected by the structure and arrangement of individual components in the system. The main contribution of this work is to develop a new analysis methodology for distributed real-time systems by reducing them to an equivalent hypothetical uniprocessor system for the purpose of analyzing the end-to-end delay of tasks.

We shall now describe the contributions made by this thesis.

1. Uniprocessor schedulability theory made great strides, in part, due to the simplicity of composing the delay of a job from the execution times of higher-priority jobs that preempt it. There is no such equivalent composition rule for distributed systems. We started by considering a very simple distributed system, namely a pipelined system, that processes several classes of real-time tasks, where each task executes on the same sequence of resources before exiting the system [38]. We derived a *delay composition rule* that allows the worst-case delay of a task invocation to be expressed in terms of the execution times of higher priority task invocations under prioritized preemptive scheduling. We showed that the end-to-end delay is bounded by that of a single virtual *bottleneck* stage plus a small additive component. This contribution effectively transformed the pipeline into a single stage system. The wealth of schedulability analysis techniques derived for uniprocessors can then be applied to decide the schedulability of the pipeline. By accurately accounting for the execution overlap between consecutive stages in the pipeline, the analysis does not become increasingly more complex and pessimistic with system scale, and significantly outperforms existing schedulability analysis techniques for distributed systems.

2. We extended the above result to non-preemptive scheduling in [40]. In uniprocessor and multiprocessor systems, preemptive scheduling always performs better than non-preemptive scheduling when the preemption overhead is zero. However, we show that in distributed systems under certain situations, non-preemptive scheduling in fact outperforms preemptive scheduling in terms of admissible system

utilization, even when the preemption overhead is assumed to be zero. That is, preemption can result in a lower system throughput than the same system under non-preemptive scheduling. This counter-intuitive property has a big impact on the nature of scheduling policies that perform well for distributed systems, making the problem of optimal distributed system scheduling extremely challenging.

3. In [42], we extended the delay composition rule to directed acyclic graphs, where different tasks can enter and leave the system at different resource stages, such that the union of all task paths forms a directed acyclic graph. We also describe how resources that are partitioned (e.g. TDMA) instead of being scheduled in a prioritized manner can be handled, by considering them as a slower resource scheduled using a prioritized scheduling policy.

4. We derived the first generalized closed form expression for schedulability analysis in distributed task systems with non-acyclic flows in [43]. Prior approaches including network calculus and holistic schedulability analysis are targeted towards acyclic task flows. They involve iterative solutions or offer no solutions at all when flows are non-acyclic. This problem of estimating the end-to-end delay of tasks in a non-acyclic system is inherently difficult due to the presence of cyclic dependencies. By considering the system as a whole rather than analyzing it one node at a time, the bound accurately accounts for the concurrency in the execution of different nodes, resulting in a less pessimistic bound on the end-to-end delay.

5. Based on the above delay composition results, we developed an algebra in [39] called *delay composition algebra*, which defines a set of simple operators for systematic transformation of distributed real-time task systems into single-resource task systems such that schedulability properties of the original system are preserved. Operands in the algebra represent workloads in composed sub-systems, and operators such as PIPE and SPLIT are applied on the operands to compose sub-systems together to reduce the distributed system to an equivalent hypothetical single resource for the purpose of schedulability analysis. In [47], we introduced the LOOP operator to handle task graphs that may contain cycles.

6. The above reduction-based approaches to schedulability transform distributed system workloads into equivalent uniprocessor workloads that can be analyzed using techniques borrowed from uniprocessor literature. However, this approach suffers from pessimism that results from mismatches between assumptions made in the uniprocessor analysis and characteristics of workloads reduced from distributed systems, especially for periodic tasks. This motivates research on uniprocessor task models that better match the peculiarities of task loads reduced from distributed systems. To address this problem, we introduce *flow-based mode changes* [44], a uniprocessor load model tuned to the novel constraints of

workloads reduced from distributed system tasks. This is the first uniprocessor task model motivated by the needs of reduction-based schedulability analysis techniques for distributed systems.

7. Large and complex distributed systems typically execute soft real-time applications, where there is significant uncertainty in the execution times of tasks on individual resources, or the worst-case timing is not entirely verified. An extremely important problem in such systems, is how do we optimize the allocation of resources to individual execution stages of tasks (the topology of the system) to minimize the effect that the uncertainties have on the end-to-end delay of tasks. In [46], we define a metric called *structural robustness* that measures the robustness of the end-to-end timing behavior of a systems task flow graph towards unexpected violations in the worst-case application execution times on individual resources. We demonstrate that by efficiently allocating resources to execution stages of end-to-end tasks, the flow paths of tasks can be optimized to improve the systems structural robustness. We also present a simple hill climbing algorithm that can be used to explore the space of all system configurations to determine a highly robust configuration.

8. The theory developed in this thesis lends itself to a wide range of applications. We adapted the delay constraints developed above to the case of wireless networks. We consider a set of elastic real-time flows in a multi-hop wireless network and consider the problem of distributed rate allocation for the flows, such that all packet deadlines are met [41]. Due to the inherent difficulty of providing hard guarantees, we formulate the problem as one of utility maximization, where the achieved utility depends on the ability to meet deadlines. Using the delay composition theorem, we relate the end-to-end delay of prioritized flows to flow rates and priorities, then impose end-to-end delay constraints that can be expressed in a decentralized manner in terms of flow information available locally at each node. The solution to the network utility maximization (NUM) problem yields a distributed rate control algorithm that nodes can independently execute to collectively maximize global network utility, subject to delay constraints.

9. We also generalize the fundamental results from delay composition theory to the case of wireless networks under arbitrary schedulability constraints [45]. In particular, given a set of flows in a wireless network with flow rates f_i and an arbitrary set of interference constraints between links in the wireless network, we obtain a scheduling policy such that the worst-case end-to-end delay of packets of flows can be upper bounded as $O(1/f_i + H_i/link_rate)$, where H_i is the number of hops in the route followed by flow i. Notice that such an upper bound in the end-to-end delay is better by a multiplicative factor, than the end-to-end delay bound of $O(H_i/f_i)$ obtained using Weighted Fair Queuing (WFQ). This

result will prove that regardless of the additional schedulability constraints, it is still possible to obtain an end-to-end delay bound that is inversely proportional to the delay on one hop only ($O(1/f_i)$), with a constant delay for every successive hop, similar to our delay composition results.

The rest of this thesis is organized as follows. In Chapter 2, we discuss related work. We describe the first delay composition result for real-time pipelines under preemptive and non-preemptive scheduling in Chapter 3. We describe the extension of this result to Directed Acyclic Graphs in Chapter 4, and to non-acyclic graphs in Chapter 5. In Chapter 6, we describe delay composition algebra. We present flow-based mode changes, the uniprocessor model with mode-changes developed for the purpose of handling workloads reduced from distributed systems in Chapter 7. We present our work on improving the structural robustness of systems towards uncertainties in worst-case execution times in Chapter 8. We discuss applications of the theory in the context of wireless networks in Chapter 9. We conclude this thesis with some directions for future research in Chapter 10.

Chapter 2

Related Work

Rigorous theory exists today for schedulability analysis of uniprocessors and multiprocessors, while mostly heuristics are used to analyze larger arbitrary-topology distributed systems. We start by reviewing some of the important work on analyzing delay and schedulability of real-time jobs in distributed systems. We then review work in the area of analyzing delay in wireless networks.

2.1 Distributed Systems

Several scheduling algorithms have been proposed for statically scheduling precedence constrained tasks in distributed systems [77, 94, 23]. Given a set of periodic tasks, such algorithms attempt to construct a schedule of length equal to the least common multiple of the task periods. The schedule will accurately specify the time intervals during which each task invocation will be executed. Needless to say, such algorithms have a large time complexity and are clearly unsuitable for large and complex distributed systems, where simplicity is of essence.

Analyzing the Worst Case Execution Times (WCET) of tasks in processor and memory pipeline architectures is a well studied problem in the area of real-time operating systems ([96, 80] and references thereof). Such algorithms execute in time that is exponential in the number of tasks in the system. Further, the approach would be difficult to implement in a distributed setting and is more error-prone.

The system model considered by us in this thesis has been studied in the context of job fair scheduling. For the case of pipelined distributed systems, polynomial-time algorithms have been proposed to construct a feasible schedule of executing the jobs under special cases where the problem is tractable, and heuristic scheduling algorithms are used to solve the general case ([9] and references thereof). In contrast, the problem we study, is to test the schedulability of a given set of priority-ordered tasks scheduled according to a given scheduling policy.

Offline schedulability tests have been proposed that divide the end-to-end deadline of tasks into per-stage deadlines. The end-to-end task is then considered as several independent sub-tasks, each executing

on a single stage in the system. Uniprocessor schedulability tests are then used to analyze if each stage is schedulable. If all the stages are schedulable, the system is deemed to be schedulable. We refer to this technique as *traditional* in our simulation studies. For instance, [50, 97] present techniques to divide the end-to-end deadline into per-stage deadlines. While this technique does not incur any problems with handling cycles in the task graph, it tends to be extremely pessimistic and does not accurately account for the inherent parallelism in the execution of different stages. A distributed pipeline framework was presented in [14], where a complex, heterogeneous, multi-resource system is decomposed into a set a single resource scheduling problems. Each single resource scheduling problem corresponds to a stage in the multi-stage pipelined distributed system.

Existing techniques to analyze distributed systems tend to become more pessimistic or offer no solutions at all for large task graphs that contains cycles. The two main techniques to analyze delay in distributed systems are holistic analysis [89] and network calculus [18, 19], and their various extensions.

Holistic analysis was first proposed in [89], and has since had several extensions such as [83, 68, 73] that propose offset-based response time analysis techniques for EDF. In addition to the computation time and period, tasks are characterized by two other parameters, namely the jitter and the offset. The jitter denotes the maximum deviation for the arrival of an invocation of a task from the periodic arrival pattern, and the offset denotes the minimum duration after which the task is activated and is ready to execute (the original holistic analysis technique [89] does not use offset information). The jitter and offset information is used to characterize the arrival pattern of tasks to each stage in the distributed system. The fundamental principle behind holistic analysis and its extensions is that, given the jitter and offset information of jobs arriving at a stage one can compute (in a worst-case manner) the jitter and offset for jobs leaving the stage, which in turn becomes the arrival pattern for jobs to a subsequent stage. By successively applying this process to each stage in the system, one can compute a worst-case bound on the end-to-end delay of jobs. However, this technique works only in the absence of cycles in the task graph. In the presence of cycles, the jitter and offset of jobs at a stage (that is part of the cycle) becomes directly or indirectly dependent on the jitter and offset of jobs leaving the stage, resulting in a cyclic dependency. To overcome this problem, an iterative procedure is described in [68, 73] which is shown to converge. This solution technique, however, becomes tedious, complicated and quite pessimistic for large task graphs with dozens of nodes.

From the networking perspective, network calculus [18, 19] was proposed to analyze the end-to-end delay of packets of flows. This was applied to the context of real-time systems, called Real-Time Calculus first presented in [87], and has since been extended to handle different system models such as [49, 91]. In approaches based on network calculus, the arrival pattern of jobs of flows is characterized by an arrival curve.

Given a service curve for a node based on the scheduling policy used, one can determine the rate at which jobs leave the node after completing execution, which in turn serves as the arrival curve for the next stage in the flow's path. For task graphs that contain cycles, we are faced with the same cyclic dependency problem. In [19], a general solution to this problem is presented by setting up a system of simultaneous equations, which becomes difficult or impossible to solve for large systems. Needless to say, there is no means by which the solution can be efficiently automated for arbitrary task graphs.

A comparison of holistic analysis and network calculus was conducted in [52], where holistic analysis was found to be less pessimistic than network calculus in general. We show in the evaluation section that both of these techniques tend to become increasingly pessimistic with system scale. In contrast, in this thesis, we derive a simple bound on the end-to-end delay of a job in terms of the computation times of higher priority jobs that can delay it. It accurately accounts for parallelism in the execution of different stages, resulting in a less pessimistic estimate of the end-to-end delay.

In order to determine the end-to-end delay of a particular task, both holistic analysis and network calculus require complete information of the entire system, and may require the whole system to be analyzed. In contrast, the analysis presented in this thesis only requires information along the path followed by the task in question, and does not need global information.

A schedulability test based on aperiodic scheduling theory was derived in [34], for fixed priority scheduling. Although this solution handles arbitrary-topology resource systems and resource blocking, it does not consider the overlap in the execution of multiple stages in the pipeline, which is a fundamental cause of pessimism. We account for this overlap in our pipeline delay composition theorem, and reduce the schedulability analysis of a multistage pipeline system to that of single stage systems. This largely increases schedulability, and the performance of the system does not become poorer with increasing number of pipeline stages.

While a lot of work has concentrated on preemptive scheduling, only a few studies have looked at non-preemptive scheduling. Complex response time analyses with exponential running time complexities are used in [95, 48, 55, 72] to analyze systems with non-preemptive scheduling. In [52], an extension to holistic analysis to account for resource blocking under non-preemptive scheduling has been presented. Network calculus-based approaches can also be used to analyze non-preemptive systems. In contrast to such techniques, we show a transformation of the distributed system under non-preemptive scheduling to an equivalent single stage system scheduled using preemptive scheduling. This allows well known uniprocessor schedulability analysis to be applied to analyze distributed systems under non-preemptive scheduling, resulting in less pessimistic analysis.

In [64], several distributed scheduling policies for jobs that follow a single cyclic path through the system

are studied. The objective is to identify policies that reduce the mean delay as well as the variance in the delay, in order to meet strict timing and buffer constraints. Unlike the system model considered in this thesis, priorities are assigned to resource buffers (each time a job revisits a resource, it is placed in a different buffer) rather than to jobs. The Last Buffer First Serve (LBFS) policy is shown to be stable and a delay bound is calculated that resembles the pipeline delay composition theorem. The bound is a sum of two terms, the first being additive over jobs and the second being additive across the resources visited, similar to our delay composition theorem. It must be noted that the bound is for the mean end-to-end delay of tasks, rather than the worst-case studied in this thesis. For systems with many job-types following different routes, stable extensions of the policies studied are also presented. A tutorial account of some results in the field are presented in [53]. Some scheduling policies of interest are also discussed. A manufacturing system with many machines and several types of parts, each requiring execution at a different prescribed sequence of machines is studied for stability properties in [54]. Manufacturing plants with additional complexities such as variable transportation times, set-up times, and parts requiring assembly or disassembly are considered in [75]. Scheduling policies are presented that ensure that the cumulative production of each part-type trails the desired production by no more than a constant, ensuring bounded buffer requirements. This class of work provides significant intuition into the kinds of scheduling policies that help reduce mean delay in distributed systems. Yet, little is known with regard to optimal scheduling policies for distributed systems, which continues to remain an open problem.

We next discuss work relating to handling unanticipated variations in the execution times of tasks in distributed systems. Feedback control has been used in [84, 63, 62], to handle variations in the execution times, where the deadline miss ratio of the tasks is the controlled variable and the CPU utilizations are the manipulated variable. In [63], an End-to-end Utilization Control algorithm (EUCON) is presented which features a feedback control loop that ensures bounded CPU utilizations and end-to-end timing guarantees in the presence of unpredictable execution times of tasks, using online performance measurements. The fluctuations in execution times are handled by varying the rates of the tasks within the system. Such online techniques that adapt the rates of tasks can be used together with our offline technique to optimize the routes of tasks to make them more robust to uncertainties in the computation times.

Resource reclaiming techniques have been proposed to deal with unpredictable task execution times [82, 12]. In [82], resources are reclaimed from tasks that complete ahead of time, which are then used to improve the performance of the system by optimizing a feasible schedule. The technique presented in [12], determines the set of rates for the different tasks that constitute the optimal system control performance under normal conditions, and when a worst-case scenario arises, adopts an overrun management algorithm that jointly

optimizes the rates of tasks as well as the task schedule. These resource reclaiming techniques typically require modifications to specific scheduling algorithms in the operating system and may be difficult or infeasible for certain applications. Where feasible, these online overrun management algorithms can be used together with our task route optimization to improve the robustness of the system towards uncertainties in the execution times.

The issue of handling uncertainties in design parameters including execution times of tasks in automobile systems, is addressed in [27]. This work adopts an approach based on info-gap decision theory to systematically analyze the robustness of various schedules by constructing the greatest horizon of uncertainty that still satisfies all the performance requirements of the system. While this work appears to be a good way to estimate the robustness of a particular schedule, it provides no intuition as to how to modify the schedule to improve its robustness. Further, the technique may prove too complex for large distributed systems.

Sensitivity analysis such as those presented in [11, 76], can be used to determine the sensitivity of the end-to-end delay of tasks to particular execution times of tasks on stages. A generalized framework of extensibility across multiple dimensions using sensitivity analysis, including, but not limited to execution times, has been proposed in [32, 33]. While these techniques give us a good understanding of how the end-to-end delay depends on individual execution times, it provides little intuition as to how to modify the system's task flow graph to reduce the sensitivity and improve the system's structural robustness. Further, when the system has a large number of resources and tasks, determining the sensitivity of each task's end-to-end deadline to each execution time in the system can be an extremely tedious process.

2.2 Wireless Networks

Over the last few years, there has been a lot of work on applying the Network Utility Maximization (NUM) framework in wireless networks and several cross-layer optimization techniques have been proposed [15, 35, 56, 16]. A tutorial on the current state of research and open research issues is presented in [59].

The problem of network resource allocation in the presence of QoS constraints including delay has been well studied in wireline networks [22, 90, 61, 28], but not in wireless networks. In [22, 90], the problem of allocating rates along a single path (or a multicast tree) is considered. This approach overlooks the contention caused by multiple intersecting flows and the intrinsic difficulties in resource allocation in such scenarios. A technique to express the end-to-end average delay of flows as a function of link rates is presented in [90], which is based on queuing theory, assuming FIFO queues and non-prioritized traffic. The study in [22] considers more general scheduling policies, but considers heuristics to partition the end-to-end QoS

requirement into per-hop requirements. Studies in [61, 28] also partition the end-to-end requirements into per-hop requirements based on link cost metrics and load-balancing objectives, but do not consider prioritized scheduling.

In [79], the problem of rate allocation in wireline networks in the presence of end-to-end bandwidth and delay requirements is studied by formulating the problem as a NUM problem. The network partitions the end-to-end delay into local per-link delays, and models links as M/G/1 queues. Expressions from queuing theory for the delay at each link is then used to compute the average delay. In contrast, in our work we directly characterize the worst case end-to-end delay of flows in terms of the rates of flows that interfere with it. The constraints make no assumption on the arrival of packets (as against Poisson arrivals assumed in [79]). Further, we consider multiple priority classes and prioritized scheduling to ensure more efficient real-time resource allocation.

Chapter 3

A Delay Composition Theorem for Real-Time Pipelines

In this chapter, we present the first main result of delay composition theory. We are concerned with pipelined systems that process several classes of real-time tasks, in which each task executes on all stages in sequence and must exit the system within a specified end-to-end latency bound. We derive a *delay composition rule* under preemptive as well as non-preemptive scheduling, that allows the worst-case delay of a task invocation to be expressed in terms of the execution times of other task invocations. According to this rule, the delay of a task in the pipeline has two components; (i) a *job-additive* component that is proportional to the sum of invocation execution times on a single stage (but is not proportional to the number of stages), and (ii) a *stage-additive* component that is proportional to the number of stages (but not the number of task invocations). Observe that this expression is better by a multiplicative factor than one that does not account for execution overlap (i.e., assumes that a task is preempted by all invocations of higher priority tasks on all stages). The delay in that last case is proportional to the *product* of the two components above as opposed to their sum. Consequently, our composition rule yields tight delay estimates that lead to good schedulability results.

Our composition rule does not make assumptions on the scheduling policy other than that it assigns the same priority to a task invocation at all stages. No assumption on periodicity of the task set is made. No assumption is made on whether different invocations of the same task have the same priority. Hence, this rule applies to static-priority scheduling (such as rate-monotonic), dynamic-priority scheduling (such as EDF) and aperiodic task scheduling alike. The simple expression of end-to-end delay computed by the aforementioned composition rule leads to a reduction of the multi-stage pipeline system to an equivalent single-stage system. Using this transformation, it becomes possible to use the wealth of existing schedulability analysis techniques on the new single-processor task set to analyze the original pipeline.

The remainder of this chapter is organized as follows. Section 3.1 briefly describes the system model. We state the delay composition theorems under preemptive and non-preemptive scheduling in Section 3.2, and develop some intuitions. In Section 3.3, this theorem is proved under the cases of preemptive and non-preemptive scheduling. Section 3.4 constructs a transformation of the pipeline into a single stage

system accounting for execution overlap among stages in the original pipeline. In Section 3.5, we provide a numerical example illustrating the transformation and schedulability analysis under EDF scheduling. In Section 3.6, we illustrate how to use single stage schedulability analyses to analyze pipelines, based on the transformation, under preemptive as well as non-preemptive scheduling. In Section 3.7, we show results of simulation experiments that demonstrate how our new transformation outperforms previous schedulability analysis when end-to-end deadlines are small. Further, we show that under certain scenarios, non-preemptive scheduling can outperform preemptive scheduling in pipelined systems.

3.1 System Model

Consider a multi-stage distributed data processing pipeline. Periodic or aperiodic tasks arrive at this system and require execution on a set of resources (such as processors)[1], each performing one stage of task execution. For the sake of deriving a general delay composition theorem, we consider individual task invocations in isolation, not to make any implicit periodicity assumptions. We call these invocations, *jobs*. In a given system, many different jobs may have the same priority (e.g., invocations of the same task in fixed-priority scheduling). However, there is typically a tie-breaking rule among such jobs (e.g., FIFO). Taking the tie-breaker into account, we can assume without loss of generality that each individual job has its own priority. This assumption will simplify the notations used in the derivations.

By definition of a pipeline, we assume that all the jobs require processing on all the stages and in the same order. The priority of each job is assumed to be the same across all the stages of the pipeline. Let the total number of stages be N. We number these stages from 1 to N, in the order visited by the jobs. Let $A_{i,j}$ be the arrival time of job J_i at stage j, where $1 \leq j \leq N$. The arrival time of the job to the entire system, called A_i, is the same as its arrival to the first stage, $A_i = A_{i,1}$. Let D_i be the end-to-end (relative) deadline of J_i. It denotes the maximum allowable latency for J_i to complete its computation in the system. Hence, J_i must exit the system by time $A_i + D_i$. The computation time of J_i at stage j, referred to as the *stage execution time*, is denoted by $C_{i,j}$, for $1 \leq j \leq N$. Finally, let $S_{i,j}$, called the *stage start time*, be the time at which J_i starts executing on a stage j, and let $F_{i,j}$, called the *stage finish time*, be the time at which J_i completes executing on stage j.

[1]While we equate a resource to a processor, the same discussion applies to other resources such as network links and disks as long as they are scheduled preemptively and in priority order. A resource pipeline can thus contain heterogeneous resources that include processing, communication and disk I/O stages.

3.2 Problem Statement

The main contribution of the work described in this chapter lies in deriving a delay composition theorem to bound the delay experienced by any job as a function of the execution times of other jobs in the pipeline, for both preemptive as well as non-preemptive scheduling. Based on certain crucial insights, we motivate why under certain circumstances where the computation times of jobs are not too dissimilar, non-preemptive scheduling can outperform preemptive scheduling in pipelined systems.

Let the job whose delay is to be estimated be J_1, without loss of generality. Let S denote the set of all jobs that have execution intervals in the pipeline between J_1's arrival and finish time (S includes J_1). Let $\bar{S} \subseteq S$ denote the set of all jobs with higher priority than J_1 and including J_1, and let $\underline{S} \subset S$ denote the set of all jobs with lower priority than J_1. Also, let the quantities $C_{i,max1}$ and $C_{i,max2}$, for any job J_i, denote its largest and second largest stage execution times, respectively. The delay composition theorem for J_1 under preemptive scheduling is stated as follows:

Preemptive Pipeline Delay Composition Theorem. *Assuming a preemptive scheduling policy with the same priorities across all stages for each job, the end-to-end delay of a job J_1 in an N-stage pipeline can be composed from the execution parameters of jobs that preempt or delay it (denoted by set $\bar{S}$) as follows:*

$$Delay(J_1) \leq \sum_{J_i \in \bar{S}} C_{eq_i} + \sum_{j=1}^{N-1} \max_{J_i \in \bar{S}}(C_{i,j}) \tag{3.1}$$

$$\begin{aligned} C_{eq_i} &= C_{i,max1} + C_{i,max2}, \quad if\ A_1 < A_i \\ &= C_{i,max1}, \quad if\ A_1 \geq A_i \end{aligned}$$

The delay composition theorem for J_1 under non-preemptive scheduling is stated as follows:

Non-preemptive Pipeline Delay Composition Theorem. *Assuming a non-preemptive scheduling policy with the same priorities across all stages for each job, the end-to-end delay of a job J_1 in an N-stage pipeline can be composed from the execution parameters of other jobs that delay it (denoted by set S) as follows:*

$$Delay(J_1) \leq \sum_{J_i \in \bar{S}} C_{i,max1} + \sum_{j=1}^{N-1} \max_{J_i \in \bar{S}}(C_{i,j}) + \sum_{j=1}^{N} \max_{J_i \in \underline{S}}(C_{i,j}) \tag{3.2}$$

Observe that, from the perspective of deriving the delay composition theorem, we are not concerned (for the moment) with how to determine set S (or $\bar{S}$). We are merely concerned with proving the fundamental property of delay composition over any such set. From the perspective of schedulability analysis, however, it is useful to estimate a worst case S to compute worst-case delay. Trivially, in the worst case, S would include

all jobs J_i whose active intervals $[A_i, A_i + D_i]$ overlap that of J_1 (i.e., overlap $[A_1, A_1 + D_1]$). This is true because a job J_i whose deadline precedes the arrival of J_1 or whose arrival is after the deadline of J_1 has no execution time intervals between J_1's arrival time and deadline (in a schedulable system), and hence cannot be part of S. The use of the delay composition theorem for schedulability analysis is further elaborated in Section 3.4. Further, observe that the delay composition theorem addresses each job independently and is not concerned with deadlines. It is therefore valid even when higher priority jobs do not meet their deadlines.

Let us for the moment concentrate on the preemptive version of the delay composition theorem. To appreciate the significance of the theorem let us consider a numeric example. Consider a set of two periodic tasks, T_1 and T_2, executing on a six-stage pipeline. Let the computation time of each task on each stage be the same and equal to 1. Let T_1 have a period of 9, equal to its *end-to-end* deadline. Let T_2 have a period of 6, also equal to its end-to-end deadline. We further assume that the first job (i.e., invocation) of each task arrives to the system at the same time. Figure 3.1 depicts the periods of invocations of the two tasks, and shows that at most two invocations of T_2 can preempt T_1. Is the task set schedulable? Assume that EDF is used on each stage.

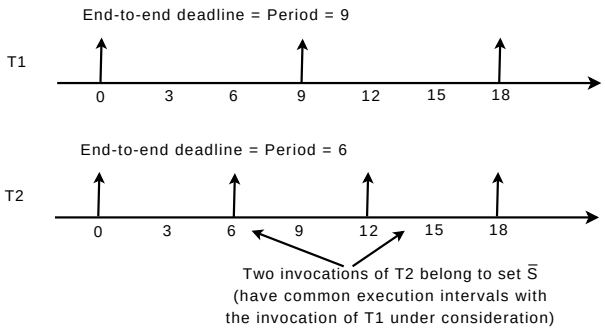

Figure 3.1: Figure illustrating example.

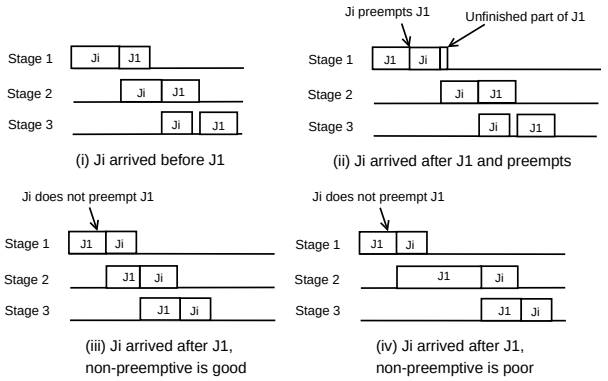

Figure 3.2: Figure showing the possible cases of two jobs in the system.

A common way to solve this problem is to partition the end-to-end deadline of each task into per-stage deadlines then analyze the schedulability of each stage independently. In this example, since the load is equal on all stages, we divide the end-to-end deadlines equally among stages, leading to a per-stage deadline of 1.5 for T_1 and 1 for T_2. Note that, T_2 has zero slack on each stage. It runs first and meets its per-stage deadlines. However, T_1 needs up to two time units to complete on a stage, which is larger than its 1.5 per-stage deadline. For T_1 to be guaranteed in this six-stage system, the above analysis requires its end-to-end deadline to be at least $2 * 6 = 12$.

Now, let us apply Equation (3.1) to calculate the delay of an invocation of T_1. Since, in this example, invocations of T_2 have a higher priority than those of T_1 and we know that any invocation of T_1 can be

preempted by at most 2 invocations of T_2 (as shown in Figure 3.1), the set $\bar{S}$, in Equation (3.1), contains only two invocations of T_2 along with the invocation of T_1 under consideration. Moreover, in any given period of T_1 only one of the two invocations of T_2 satisfies $A_1 < A_2$ (leading to $C_{eq_2} = 2$ for one invocation and 1 for the other). $C_{eq_1} = 1$. Hence, the first summation is equal to $2 + 1 + 1 = 4$. The second summation adds 5 leading to a total delay of 9 for T_1. This is lower than 12 above and does not exceed T_1's end-to-end deadline. The system is found schedulable. In other words, our results can lead to less pessimistic pipeline schedulability analysis. The explanation is as follows.

The traditional analysis (i.e., breaking the end-to-end deadline into per-stage deadlines and performing a single stage schedulability test) is pessimistic because it assumes a worst-case arrival pattern. In other words, it assumes that an invocation of T_1 and T_2 arrive together, leading to a delay of 2 for T_1. In reality, this is not true of each stage. For example, if this arrival pattern was true at the first stage, T_2 would execute ahead of T_1 on that stage and move on to the next. From then on, T_1 would execute on each stage n concurrently with the execution of T_2 on stage $n + 1$. T_1 would never wait for T_2 again, since every time T_1 would advance to the next stage, T_2 would leave it to the one after. It is important to account for this execution overlap. Indeed, if T_1 and T_2 start together, T_1 will take 2 time units on the first stage and one of each subsequent stage, finishing in only 7 time units.[2] Clearly, need arises to better account for the effect of pipelining and execution overlap, which is what we purport to do.

The following question might then arise: is the common practice of partitioning end-to-end deadlines into per-stage deadlines always pessimistic? The answer is no. For example, consider a task set with *per-stage* deadlines equal to their periods. The set is schedulable using EDF at up to 100% utilization on each machine. There is no room for improvement in this case. The difference between this and the previous example lies in the ratio of task end-to-end deadlines to periods. In the current example, this ratio is equal to the number of stages. In the previous example this ratio was 1. While the results of this work are general, they offer improvement over the state of the art only in the case where the ratio of end-to-end deadlines to periods of tasks is sufficiently smaller than the number of stages. In particular, the theory offers great improvements for aperiodic tasks (where periods are "infinite" and hence satisfy the above condition).

Let us now compare the forms of the two delay composition theorems. The first term in the delay bound expression under both theorems is a summation over all higher priority jobs, and is termed the *job-additive* component of J_1's delay. Notice that the preemptive version considers two maximum stage execution times of each higher priority job that arrives after J_1, while the non-preemptive version considers just one maximum stage execution time. The reason for this will be explained shortly. The second term in both cases is a

[2]We show later that this is not the worst case scenario, but the system is indeed schedulable.

summation over all stages and is independent of the number of jobs in the system. Further, one maximum stage computation time of a lower priority job is added at each stage to account for blocking under non-preemptive scheduling (in the worst case J_1 will be blocked by a lower priority job at each stage). The second and third terms in the non-preemptive case, and the second term in the preemptive case, is called the *stage-additive* component of J_1's delay.

Finally, it is interesting to note that preemption in pipelines can reduce execution overlap among stages (which explains why C_{eq_i}, in the preemptive delay composition rule, depends on which job comes first). For example, consider the case of a two-job pipeline system shown in Figure 3.2. In Figure 3.2(i), the higher-priority job J_i arrives together with J_1 and is given the (first-stage) CPU. When J_i moves on to the second stage, J_1 can execute in parallel on the first. However, as shown in Figure 3.2(ii), if J_i arrives *after* J_1 and preempts it, when J_i moves on to the next stage, only the *unfinished* part of J_1 on the stage where it was preempted can overlap with J_1's execution on the next stage. In other words, execution overlap is reduced and J_1 takes longer to finish than it did in the previous case. This is the reason why under preemptive scheduling, two maximum stage execution times need to be considered for each higher priority job that arrives after J_1 to the system. For instance, in our six-stage example, presented above, the aforementioned arrival scenario gives an actual delay of 8, not 7, for T_1. Now let us consider the execution of the two jobs under non-preemptive scheduling, shown in Figure 3.2(iii) for the same arrival times as in Figure 3.2(ii). Notice that job J_1 finishes much earlier under non-preemptive scheduling, and J_i is only marginally delayed. Thus, the system can sustain a greater load (throughput) under non-preemptive scheduling. However, this observation that non-preemptive scheduling can perform better than preemptive scheduling for distributed systems, is true only when the execution times of jobs are relatively similar. For example, Figure 3.2(iv) illustrates a scenario where the higher priority job J_i is blocked for a significantly long duration, waiting for the lower priority job J_1 to complete execution, resulting in J_i to possibly miss its deadline. This shows that there are instances where non-preemptive scheduling can outperform preemptive scheduling in distributed systems, and instances where the opposite is true. It would be interesting to mathematically quantify the scenarios under which one will perform better than the other. In Section 3.7, we characterize through simulations the space in which non-preemptive scheduling can perform better than preemptive scheduling.

With the intuitions explained above, we now prove the pipeline delay composition theorems under preemptive and non-preemptive scheduling. In the proof below, we consider individual jobs and not tasks in order to be general. By considering jobs we do not restrict the results to the special case of periodic arrivals.

3.3 Delay Composition for Pipelined Systems

We first prove the preemptive version of the delay composition theorem in Section 3.3.1. For the sake of brevity, we only show how this proof can be extended to the non-preemptive version in Section 3.3.2.

3.3.1 Proof for the Preemptive Case

The delay composition theorem can be proved by induction on task priority. We first prove the theorem for a two-job scenario (Lemma 1). We then prove the induction step, where we assume that the delay composition theorem is true for $k-1$ jobs, $k \geq 3$, add a k^{th} job with highest priority, and prove that the delay composition theorem still holds.

Lemma 1. *When J_1 and J_2 are the only two jobs in the system, and J_2 has a higher priority than J_1, the delay experienced by J_1 is at most*

$$Q = \sum_{i=1}^{2} C_{eq_i} + \sum_{j=1}^{N-1} max_{i=1,2}(C_{i,j}), \tag{3.3}$$

where:

$$\begin{aligned} C_{eq_i} &= C_{i,max1} + C_{i,max2}, \quad if\ A_1 < A_i \\ &= C_{i,max1}, \quad if\ A_1 \geq A_i \end{aligned}$$

Proof. We shall prove the lemma by considering two cases; J_2 arrived before (or together with) J_1, and J_2 arrived after J_1 (special cases where one task arrives after the other exits the system can be trivially shown to satisfy the lemma as well).

Case 1: J_2 arrived before or together with J_1 to the system

Since J_2 is the highest-priority job in the system, it executes uninterrupted on all stages, completing each stage k exactly after a time equal to $C_{2,1} + ... + C_{2,k}$. Job J_1 executes after J_2 on the first stage. When J_1 finishes some stage, it moves to the next, where it may encounter J_2 (again) and must wait for it to finish. If J_2 had already cleared that stage, J_1 can execute there immediately. Let stage L be the last stage where J_1 had to wait for J_2. In this case, as shown in Figure 3.3-a, J_1 completes the pipeline with a delay at most equal to:

$$Delay(J_1) \leq C_{2,1} + ... + C_{2,L} + C_{1,L} + ... + C_{1,N} \tag{3.4}$$

Note that, $C_{2,1} + ... + C_{2,L}$ takes us to the completion time of J_2 on stage L (where J_1 last waited for J_2). $C_{1,L} + ... + C_{1,N}$ is the additional time taken by J_1 to execute on L and the remaining stages. The delay expression in Inequality (3.4) has $N+1$ terms, each representing a per-stage job computation time. There is exactly one per-stage computation in this expression from each stage, except stage L that contributes two.

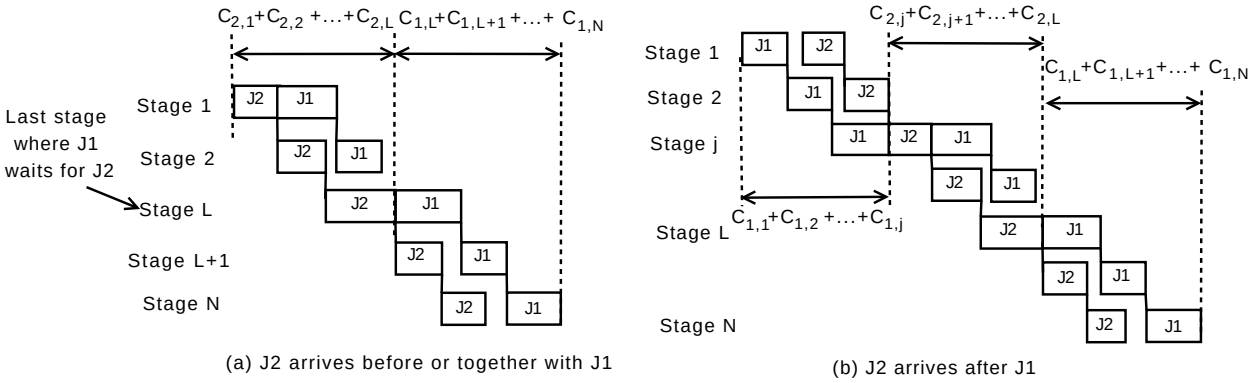

Figure 3.3: Figure showing the delay for the two cases of Lemma 1.

To compute a delay bound, let us replace one per-stage computation time at each of the first $N-1$ stages by $\max_{i=1,2}(C_{i,j})$ for that stage. Inequality 3.4 can now be re-written as:

$$Delay(J_1) \leq \sum_{j=1}^{N-1} max_{i=1,2}(C_{i,j}) + C_{2,L} + C_{1,N} \tag{3.5}$$

Since the last two terms are at most $C_{eq_2} = C_{2,max1}$ and $C_{eq_1} = C_{1,max1}$ respectively, the expression in the lemma is a valid upper bound.

Case 2: J_2 arrived after J_1 to the system

Let J_2 preempt J_1 on some stage j. Up to stage $j-1$, the delay of J_1 on each stage is simply its own execution time. At stage j, J_2 preempts J_1 after the latter has executed for some time $C^*_{1,j} < C_{1,j}$. As in the case above, J_1 executes after J_2 on subsequent stages. Let L be the last stage where J_1 waits for J_2. The delay of J_1 is thus $C_{1,1} + ... + C^*_{1,j} + C_{2,j} + ... + C_{2,L} + C_{1,L} + ... + C_{1,N}$, as shown in Figure 3.3-b. Following the same substitution as above, we can show that:

$$Delay(J_1) \leq \sum_{j=1}^{N-1} \max_{i=1,2}(C_{i,j}) + C_{2,j} + C_{2,L} + C_{1,N} \tag{3.6}$$

Since $C_{2,j} + C_{2,L} \leq C_{eq_2}$ and $C_{1,N} \leq C_{eq_1}$, the expression in the lemma is a valid upper bound in this case as well. This completes the proof of the lemma. □

We now prove the pipeline delay composition theorem by induction on job priority.

Preemptive Pipeline Delay Composition Theorem. *Assuming a preemptive scheduling policy with the same priorities across all stages for each job, the end-to-end delay of a job J_1 of lowest priority in an N-stage pipeline with $n-1$ higher priority jobs is at most*

$$Delay(J_1) \leq \sum_{i=1}^{n} C_{eq_i} + \sum_{j=1}^{N-1} \max_{i=1}^{n}(C_{i,j})$$

where C_{eq_i} is as defined in Lemma 1.

Proof. Without loss of generality, we assume that a job J_i has a higher priority than a job J_k, if $i > k, i, k \leq n$. That is, J_n has the highest priority, and J_1 has the least priority.

The basis step is the case when there are only two jobs in the system, J_1 and J_2. The delay composition theorem for two jobs is precisely Lemma 1.

Assume that the result is true for $n = k - 1$ jobs, $k \geq 3$. That is,

$$Delay_{k-1}(J_1) \leq \sum_{i=1}^{k-1} C_{eq_i} + \sum_{j=1}^{N-1} \max_{i \leq k-1} (C_{i,j}) \tag{3.7}$$

We need to show the result when a k^{th} job J_k, with highest priority, is added. Let L_k be a pipelined system with k jobs, with arbitrary arrival times for each of the jobs. Let L_{k-1} be the system without job J_k. The outline of the proof is similar to the proof of Lemma 1. We consider two cases, J_k arrived before (or together with) J_1 to the system, and J_k arrived after J_1 to the system.

Case 1: J_k arrived before or together with J_1 to system L_k.

Notice that, if there exists an idle time between the execution of J_k and J_1 on some stage j, the delay of J_1 on stage j is independent of the execution time of J_k (and other jobs that execute before the idle time) on stage j. Therefore, beyond the last stage j, where there is no idle time between the execution of J_k and J_1, J_k will not influence the delay of J_1 (jobs that J_k preempts on a stage will also execute before the idle period on that stage). After J_k completes execution on stage j, the delay of J_1 in system L_k is at most its worst case delay in system L_{k-1} starting from stage j. As we make no assumption on the arrival pattern of higher priority jobs, the delay composition theorem provides the worst case delay for any possible arrival pattern of jobs. Although, adding job J_k does perturb the schedule, the worst case delay due to jobs J_2 through J_{k-1} as per the delay composition theorem accounts for any arrival pattern of J_2 through J_{k-1}. We can therefore apply induction assumption starting from stage j. Hence, the delay of J_1 can be expressed as the delay up to the time J_k completes execution on stage j ($F_{k,j}$), added to the worst case delay of J_1 in system L_{k-1} starting from stage j (as shown in Equation 3.8). This is shown in Figure 3.4.

$$\begin{aligned} Delay_k(J_1) &= F_{1,N} - A_{1,1} \\ &= (F_{1,N} - F_{k,j}) + (F_{k,j} - A_{1,1}) \end{aligned} \tag{3.8}$$

As J_k arrived before J_1 to the system, the duration between the arrival of J_1 to the system ($A_{1,1}$) and the completion of J_k's execution on stage j, is at most the time J_k takes to complete execution up to stage j

$(F_{k,j} - A_{k,1})$ (Inequality 3.9). J_k is the highest priority job in the system, and does not wait to execute on any of the stages. The time for J_k to complete execution up to stage j is $(\sum_{t=1}^{j-1} C_{k,t} + C_{k,j})$. In addition to this, from induction assumption, the delay of J_1 from stages j through N is $\sum_{i=1}^{k-1} C_{eq_i} + \sum_{t=j}^{N-1} \max_{i \leq k-1}(C_{i,t})$ (Inequality 3.10). Thus,

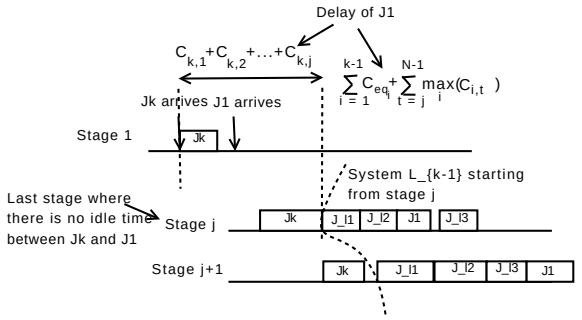

Figure 3.4: Figure showing the delay of J_1 for the case when J_k arrived before J_1.

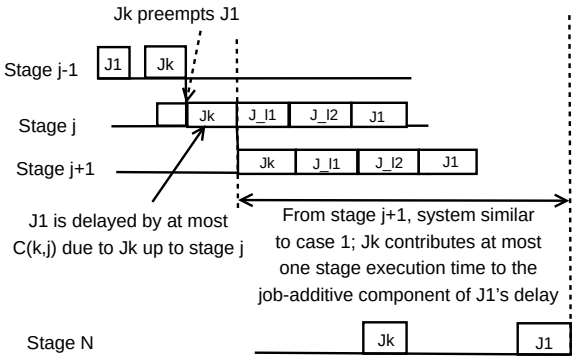

Figure 3.5: Figure showing the case when J_k arrived after J_1 and preempts J_1 at stage j.

$$
\begin{aligned}
Delay_k(J_1) &\leq (F_{1,N} - F_{k,j}) + (F_{k,j} - A_{1,1}) \\
&\leq (F_{1,N} - F_{k,j}) + (F_{k,j} - A_{k,1}),\ as\ A_{k,1} < A_{1,1} && (3.9) \\
&\leq (\sum_{t=1}^{j-1} C_{k,t}) + C_{k,j} + \sum_{i=1}^{k-1} C_{eq_i} + \sum_{t=j}^{N-1} \max_{i \leq k-1}(C_{i,t}) && (3.10) \\
&\leq \sum_{i=1}^{k} C_{eq_i} + \sum_{t=1}^{N-1} \max_{i \leq k}(C_{i,t}) && (3.11)
\end{aligned}
$$

which proves the delay composition theorem.

Case 2: J_k arrived after J_1 to the system.

Until the time J_k preempts J_1, the delay of J_1 is independent of J_k. Let J_k preempt J_1 at stage j. Beyond stage j, J_k arrives at each stage before J_1. Therefore, the pipeline beyond stage j can be thought of as one having $N - j$ stages, and J_k arriving before J_1. We can then apply the result from case 1.

The fact that J_k preempted some job at stage j (it is possible that J_k preempted some job, which in turn had preempted J_1), implies that there was a job executing when J_k arrived at stage j. Further, there is no idle time between the executions of J_k and J_1. Let J_{l_1}, J_{l_2}, ..., J_{l_s}, be the jobs that execute between J_k and J_1 on stage j (Figure 3.5). J_{l_1} is delayed by J_k up to stage j by at most $C_{k,j}$. Similarly, irrespective of previous stages, each of J_{l_2}, J_{l_3}, ..., J_{l_s}, and J_1 are delayed by an amount $C_{k,j}$ due to J_k up to stage j.

Beyond stage j, as mentioned earlier the system is identical to case 1 (as J_k arrived before J_1 to stage $j+1$). From the result of case 1, the additional delay that J_k causes J_1 is one maximum stage execution time between stages $j+1$ through N, apart from J_k's contribution to the stage-additive component $\max_i(C_{i,t})$,

for $j+1 \leq t \leq N-1$ (from Inequality 3.10). Figure 3.5 shows this scenario. We showed that the delay due to J_k up to stage j is at most $C_{k,j}$. Therefore, the total job-additive delay to J_1 due to J_k is at most the sum of the two maximum stage execution times of J_k, that is $C_{k,max1} + C_{k,max2} = C_{eq_k}$.

This proves the induction step. Using this together with Lemma 1, the theorem is proved. □

3.3.2 Proof for the Non-Preemptive Case

We first prove the delay composition theorem in the presence of higher priority jobs alone. As this is similar to the preemptive case, for the sake of brevity, we only outline this proof in Lemma 2. We then show how the presence of lower priority jobs can be accounted for, and show that the delay due to resource blocking in only proportional to the number of stages and is independent of the number of lower priority jobs.

Lemma 2. *Assuming a non-preemptive scheduling policy with the same priorities across all stages for each job, the end-to-end delay of a job J_1 of lowest priority in a pipeline with $n-1$ higher priority jobs is at most*

$$Delay(J_1) \leq \sum_{i=1}^{n} C_{i,max} + \sum_{t \leq N-1} \max_{i=1}^{n}(C_{i,t})$$

Proof. The proof of this lemma is very similar to the proof of the preemptive case (Section 3.3.1). However, there is one main difference which needs to be carried forth throughout the proof. As motivated in Section 3.1, while two maximum stage execution times of higher priority jobs are considered under preemptive scheduling, a higher priority job can overtake J_1 at most once, and can hence delay J_1 by at most one maximum stage execution time under non-preemptive scheduling. The stage-additive component is the sum of one maximum stage execution time of any job accrued over all stages. □

We now proceed to characterize the delay due to lower priority jobs and prove the theorem in its entirety.

Non-preemptive Pipeline Delay Composition Theorem. *Assuming a non-preemptive scheduling policy with the same priorities across all stages for each job, the end-to-end delay of a job J_1 in an N-stage pipeline can be composed from the execution parameters of other jobs that delay it (denoted by set S) as follows:*

$$Delay(J_1) \quad \leq \quad \sum_{J_i \in \bar{S}} C_{i,max} + \sum_{j \leq N-1} \max_{J_i \in \bar{S}}(C_{i,j}) + \sum_{j \leq N} \max_{J_i \in S}(C_{i,j}) \tag{3.12}$$

Proof. Under preemptive scheduling lower priority jobs cause no delay to higher priority jobs. However, under non-preemptive scheduling, a higher priority job may block on a resource while a lower priority job is accessing it. In the worst case, a higher priority job may be delayed by at most one lower priority job at every stage in the pipeline. Figure 3.6 illustrates such a scenario.

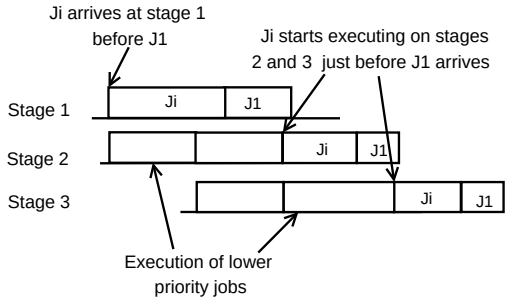

Figure 3.6: Figure illustrating how J_1 can be delayed by one lower priority job at each stage of the pipeline.

In the worst case, the lower priority job J_i would start executing at stage j, just before J_1 arrives at the stage, causing J_1 to wait for one complete stage execution time of J_i. Figure 3.6 illustrates such a scenario, where lower priority jobs delay the execution of J_i until just prior to the arrival of J_1 to stages 2 and 3. Note that, J_1 may be delayed by a different lower priority job at each stage of the pipeline. Thus, J_1 is delayed by at most one maximum stage execution time of any lower priority job at each stage which is the third term in the delay bound as per the delay composition theorem. This delay is in addition to J_1's own computation times on each of the N stages, which is accounted as one maximum stage execution time on each of the first $N-1$ stages (the second term in the delay expression), and one maximum stage execution time of J_1 (part of the first term).

In the proofs of Lemma 2, we assumed a worst case arrival pattern of higher priority jobs that cause a worst case delay to job J_1. This worst case arrival pattern of each higher priority job is independent of other jobs in the system, and is therefore applicable in the presence of lower priority jobs too. A detailed proof is omitted in the interest of brevity. This completes the proof sketch of the delay composition theorem for the non-preemptive case. □

3.4 Schedulability and Pipeline Reduction

In this section, we illustrate a systematic reduction of the pipeline schedulability problem to an equivalent single stage problem using the delay composition theorem. Since delay predicted by the delay composition theorem grows with set S, let us first define the *worst-case* (i.e., largest) set S, denoted S_{wc}, of jobs that delay or preempt J_1. In this work, we suggest a very simple (and somewhat conservative) definition of set S_{wc}. We expect that future work can improve upon this definition using more in-depth analysis. In the absence of further information, set S_{wc} is defined as follows.

Definition: The worst-case set S_{wc} of jobs that delay or preempt job J_1 (hence, include execution intervals between the arrival and finish time of J_1) includes all jobs J_i whose intervals $[A_i, A_i + D_i]$ overlap the interval

where J_1 was present in the pipeline, $[A_1, A_1 + Delay(J_1)]$.

Observe that the above is a conservative definition. It simply excludes the impossible. In a schedulable system, a job J_i that does not satisfy the above condition either completes prior to the the arrival of J_1 or arrives after its completion. Hence, it cannot possibly have execution intervals that delay or preempt J_1. Let $\bar{S}_{wc} \subseteq S_{wc}$ denote the set of all jobs with higher priority than J_1 and including J_1, and let $\underline{S}_{wc} \subset S_{wc}$ denote the set of all jobs with lower priority than J_1. In Sections 3.4.1 and 3.4.2, we show how the reduction of the pipeline to a single stage is carried out to analyze the schedulability of each task in the original system under preemptive and non-preemptive scheduling, respectively.

3.4.1 Reduction of Pipeline to an Equivalent Single Stage Under Preemptive Scheduling

Let us divide the set $\bar{S}_{wc}$ into the subset $\bar{S}_{bef} \subseteq \bar{S}_{wc}$ that contains those jobs with $A_i \leq A_1$, and a subset $\bar{S}_{after} \subset \bar{S}_{wc}$ that contains those jobs with $A_i > A_1$. We can now rewrite the delay composition theorem, separating its first summation into two; one for invocations that arrive before (or with) T_1, and one for those that arrive after. This allows us to substitute for C_{eq_i} accordingly in each summation, resulting in the following:

$$Delay(J_1) \leq \sum_{J_i \in \bar{S}_{bef}} C_{i,max1} + \sum_{J_i \in \bar{S}_{after}} (C_{i,max1} + C_{i,max2}) + \sum_{j=1}^{N-1} \max_i(C_{i,j}) \tag{3.13}$$

The reduction to a single stage system under preemptive scheduling is then conducted by (i) replacing each pipeline job J_i in $\bar{S}_{bef}$ by an equivalent single stage job (with the same priority and deadline) of execution time equal to $C_{i,max1}$, (ii) replacing each pipeline job J_i in $\bar{S}_{after}$ by an equivalent single stage job of execution time equal to $C_{i,max1} + C_{i,max2}$, and (iii) adding a lowest-priority job, J_e^* of execution time equal to $\sum_{j=1}^{N-1} \max_i(C_{i,j})$ (which is the last term in Inequality (3.13)), and deadline same as that of J_1. By the delay composition theorem, the total delay incurred by J_1 in the pipeline is no larger than the delay of J_e^* on the uniprocessor, since the latter adds up to the delay bound expressed on the right hand of Inequality (3.13).

For example, let us illustrate this transformation in the case of rate-monotonic scheduling of periodic tasks with periods equal to *end-to-end* deadlines. Consider a set of periodic tasks arriving at a pipeline, where each task T_i has a period P_i. As shown in Figure 3.7, there can be at most one invocation of each higher-priority task T_i in set $\bar{S}_{bef}$. Similarly, the number of invocations of each task T_i that arrive after the invocation of T_1 (say J_1) and delay it, is no larger than $\lceil \frac{Delay(J_1)}{P_i} \rceil$. Following the reduction outlined above, then aggregating jobs of the same period into single periodic tasks, the following periodic task set is reached:

- Task T_e^* (of lowest priority), with a computation time

$C_e^* = \sum_{J_i \in \bar{S}_{bef}} C_{i,max1} + \sum_{j=1}^{N-1} \max_i(C_{i,j})$. The task further has the same period and deadline as T_1 in the original set.

- Tasks T_i^*, each has the same period and deadline as one T_i in the original set, and has an execution time equal to $C_i^* = C_{i,max1} + C_{i,max2}$.

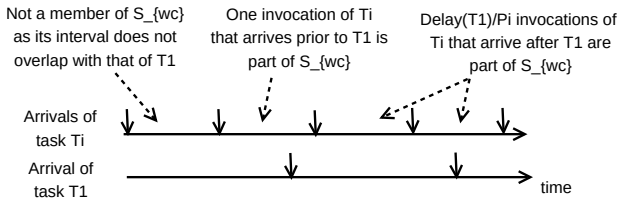

Figure 3.7: Invocations in S_{wc}.

Hence, if task T_e^* is schedulable on a uniprocessor, so is J_1 on the original pipeline. The transformation is complete. In Section 3.6, we present pipeline schedulability expressions for deadline monotonic scheduling based on the above task set reduction.

Likewise, the transformation to analyze the schedulability of T_1 in the pipeline, for the same task set under EDF scheduling, results in the following single stage task set (the set S_{bef} includes all invocations of tasks that have an earlier arrival time to the system and higher priority than the invocation of T_1 under consideration):

- Task T_e^* with a computation time $C_e^* = \sum_{J_i \in S_{bef}} C_{i,max1}$

  $+ \sum_{j=1}^{N-1} \max_i(C_{i,j})$. Note that under EDF one invocation of every other task can have an earlier arrival time and end-to-end deadline (and higher priority) than that of the invocation of T_1 (and hence be part of S_{bef}). This is in contrast to RM scheduling, where all invocations of a task have the same priority, and only invocations of higher priority tasks were part of S_{bef}. T_e^* further has the same period and deadline as T_1 in the original set.

- For each T_i that has a smaller relative deadline than T_1 in the original task set, there is a corresponding task T_i^*, having the same period and deadline as T_i and execution time equal to $C_i^* = C_{i,max1} + C_{i,max2}$. This is due to the fact that only tasks with a smaller relative deadline than T_1 can arrive after the invocation of T_1 and have an earlier absolute deadline.

In Section 3.5, we provide a numeric example to illustrate the transformation and schedulability analysis of a periodic task set under EDF scheduling.

3.4.2 Reduction of Pipeline to an Equivalent Single Stage Under Non-Preemptive Scheduling

In this section, we show how the schedulability analysis of a job in a pipeline scheduled using non-preemptive scheduling can be reduced to that in an equivalent single stage system under *preemptive* scheduling. This is performed by (i) replacing each job J_i in $\bar{S}_{wc}$ by an equivalent single stage job of execution time equal to $C_{i,max}$, and (ii) adding a lowest-priority job, J_e^* of execution time $\sum_{j \leq N-1} \max_{J_i \in \bar{S}_{wc}}(C_{i,j}) + \sum_{j \leq N} \max_{J_i \in \underline{S}_{wc}}(C_{i,j})$ (which are the last two terms in Inequality (3.2)), and deadline same as that of J_1. The delay due to blocking of resources by lower priority jobs is included as part of the execution time of J_e^*. Further, note that in the above transformation, the constructed single stage system is scheduled using preemptive scheduling, while the original pipeline system was scheduled using non-preemptive scheduling. This is due to the fact that higher priority jobs can overtake J_1 in the pipeline, which corresponds to the equivalent higher priority jobs preempting J_1 in the single stage system. It follows from the non-preemptive delay composition theorem, that the end-to-end delay experienced by J_1 in the pipelined system under non-preemptive scheduling, is no larger than the delay experienced by J_e^* in the uniprocessor system under preemptive scheduling.

Thus, if J_e^* is schedulable on the uniprocessor, so is J_1 on the original pipelined system. The reduction for periodic tasks can be conducted similar to the description under preemptive scheduling. In Section 3.6, we show how well known uniprocessor schedulability analysis can be applied to the analysis of pipelines scheduled under non-preemptive scheduling.

3.5 A Numeric Example

To illustrate the application of the above approach, consider a three stage pipeline traversed by four tasks whose per-stage computation times, end-to-end deadlines, and periods are given in Table 3.1. Let the pipeline be scheduled in an EDF manner. We assume a simple uniprocessor schedulability test that checks if the sum of the ratios of computation times to deadlines of tasks is at most 1. This test is only a sufficient test when deadlines can be lesser than the periods of tasks. In this section, we demonstrate how this task set is transformed into single-stage schedulability problems and solved to determine if the pipeline is schedulable.

Task	$C_{i,1}$	$C_{i,2}$	$C_{i,3}$	Deadline	Period
T_1	1	0.5	0.5	8	8
T_2	0.5	1	1	10	12
T_3	0.5	0.5	1	10	15
T_4	1	1	0.5	12	15

Table 3.1: Task parameters used in the example.

As there are four tasks in the system, four single stage systems need to be analyzed. While considering the schedulability of task T_i, a task $T_e^*(T_i)$ is created in the corresponding hypothetical single stage system. As the scheduling policy is EDF, $T_e^*(T_i)$'s execution time is one maximum stage execution time of every task (one invocation of every other task could be present in the system when T_i arrives) added to the maximum stage execution times on the first two stages, for all i. Therefore, $C_e^*(T_i) = 4 + 2 = 6$ time units, for all i. The deadline of $T_e^*(T_i)$ is the same as that of T_i. The other tasks that would be created in the four single stage systems are T_1^*, T_2^*, T_3^*, and T_4^*. The execution time of task T_i^* is the sum of the two largest stage execution times of task T_i, and the deadline of T_i^* is same as the deadline of T_i. Therefore, $C_1^* = 1.5$, $C_2^* = 2$, $C_3^* = 1.5$, and $C_4^* = 2$.

Task T_1: The corresponding single stage system will contain only the task $T_e^*(T_1)$ (as invocations of other tasks that arrive after $T_e^*(T_1)$ would not execute ahead of it under EDF). By applying the uniprocessor test, as $\frac{6}{8} < 1$, T_1 is schedulable.

Task T_2: The single stage system consists of two tasks $T_e^*(T_2)$ and T_1^*. As $\frac{1.5}{8} + \frac{6}{10} = 0.7875 < 1$, T_2 is schedulable.

Task T_3: The single stage system consists of three tasks $T_e^*(T_3)$, T_1^*, and T_2^*. As $\frac{1.5}{8} + \frac{2}{10} + \frac{6}{10} = 0.9875 < 1$, T_3 is schedulable.

Task T_4: The single stage system consists of four tasks $T_e^*(T_4)$, T_1^*, T_2^*, and T_3^*. $\frac{1.5}{8} + \frac{2}{10} + \frac{2}{10} + \frac{6}{12} = 1.0875 > 1$. Therefore, T_4 may not be schedulable.

However, T_4's schedulability can be analyzed by changing the schedulability test used. For example, by calculating the actual number of invocations of T_1^*, T_2^*, and T_3^* that arrive after the invocation of $T_e^*(T_4)$ and preempt it, T_4's schedulability can be precisely determined. Under EDF, only one invocation each of T_1^*, T_2^*, and T_3^* would arrive after $T_e^*(T_4)$, and have a deadline earlier than $T_e^*(T_4)$. Therefore, the total worst case delay to the invocation of $T_e^*(T_4)$ (and hence of T_4), is $C_e^*(T_4) + C_1^* + C_2^* + C_3^* = 6 + 1.5 + 2 + 1.5 = 11 < 12$. Therefore, T_4 is schedulable.

The delay composition theorem and the reduction can thus be used for a variety of scheduling policies and schedulability tests.

3.6 Utility of Derived Result

The reduction of the analysis of a multistage system to that of single stage systems based on the two delay composition theorems, enables the use of a wide range of single stage schedulability analyses, including the well known Liu and Layland bound [60], the hyperbolic bound [10], and exact tests [8, 57], to test the schedulability of tasks in multistage pipelined distributed systems. In this respect, this is indeed a 'meta-

schedulability test'. In fact, any single stage schedulability test can be used in the analysis of the multistage pipeline as long as the underlying scheduling model is prioritized scheduling, and tasks do not block for resources (except due to lower priority tasks under non-preemptive scheduling) on any of the stages (i.e., independent tasks). In the rest of this section, we concern ourselves with schedulability analysis for periodic tasks under preemptive and non-preemptive deadline monotonic scheduling. We assume that task T_i has a higher priority than task T_k, if $i < k$.

As examples, we show how the Liu and Layland bound [60] and the necessary and sufficient test based on response time analysis [8] can be applied to analyze periodic tasks in a multistage pipeline. Other uniprocessor schedulability tests can be adapted to analyze pipelines in a similar manner.

Let M be the number of periodic tasks in the system. Let $C_{i,max1}$ and $C_{i,max2}$ are the largest and second largest stage execution times of T_i, and let D_i be its end-to-end deadline. Under preemptive scheduling, let $C_e^*(i) = \sum_{k=1}^{i} C_{k,max1} + \sum_{j=1}^{N-1} \max_{k \leq i}(C_{k,j})$ and $C_k^* = C_{k,max1} + C_{k,max2}$. For non-preemptive scheduling, $C_k^* = C_{k,max1}$; $C_e^*(i) = \sum_{k \leq i} C_{k,max1} + \sum_{j \leq N-1} \max_{k \leq i}(C_{k,j}) + \sum_{j \leq N} \max_{k > i}(C_{k,j})$.

The Liu and Layland bound [60], applied to periodic tasks in a multistage pipeline is:

$$\frac{C_e^*(i)}{D_i} + \sum_{k=1}^{i-1} \frac{C_k^*}{D_k} \leq i(2^{\frac{1}{i}} - 1)$$

for each $i, 1 \leq i \leq M$. The time complexity of this analysis is $O(MN)$, as M tests have to be performed (one for each task), each of which has an $O(N)$ complexity. This complexity analysis assumes that the values $X_i = \sum_{k=1}^{i} C_{k,max1}$ and $Y_i = \sum_{k=1}^{i-1} \frac{C_k^*}{D_k}$ are stored when performing the test for task T_i. Using X_i and Y_i, X_{i+1} and Y_{i+1} can be computed in O(1) time and used in the test for task T_{i+1}, for $1 \leq i \leq M-1$.

The necessary and sufficient test for schedulability of periodic tasks under deadline monotonic scheduling proposed in [8], used together with our meta-schedulability test, will have the following recursive formula for the worst case response time R_i of task T_i:

$$\begin{aligned} R_i^{(0)} &= C_e^*(i) \\ R_i^{(k)} &= C_e^*(i) + \sum_{j<i} \left\lceil \frac{R_i^{(k-1)}}{P_j} \right\rceil C_j^* \end{aligned}$$

The worst case response time for task T_i is given by the value of $R_i^{(k)}$, such that $R_i^{(k)} = R_i^{(k-1)}$.

3.7 Simulation Results

To evaluate the actual performance of our delay composition rule and reduction to a single stage system, we constructed a simulator that models a distributed pipelined system. In order to maintain real-time

guarantees within the system, an admission controller is used. For periodic tasks, the admission controller is based on a single stage schedulability test for deadline monotonic scheduling, such as the Liu and Layland bound [60] or response time analysis [8], together with our reduction of the multistage system to a single stage, as shown in Section 3.6. When a task arrives at the system, it is tentatively added to the set of tasks in the system. The admission controller then tests whether the new task set is schedulable. The new task is admitted if the task set is schedulable, and dropped if not. Although the simulation parameters assumed in the evaluation do not reflect any realistic application, the range of values used serve as micro-benchmarks to evaluate the performance of the admission controller.

In the rest of this section, we use the term utilization to refer to the average per-stage utilization. Each point in the figures below represent average values obtained from 100 executions of the simulator, with each execution running for 30000 task invocations. Each admission controller was allowed to execute on the same 100 task sets. *End-to-end* deadlines (equal to the periods) of tasks are chosen as $10^x a$ simulation seconds, where x is uniformly varying between 0 and DR (deadline ratio parameter), and $a = 500 * N$, where N is the number of stages in the system. Such a choice of deadlines enables the ratio of the longest task deadline to the shortest task deadline to be as large as 10^{DR}. If DR is chosen close to zero, tasks would have similar deadlines. If DR is higher (for example $DR = 3$), deadlines of tasks would differ more widely. As will be demonstrated later in this section, we observed from our simulations that the achievable utilization varied significantly with the choice of DR. The default value for DR is taken to be 1. The execution time for each task on each stage was chosen based on the task resolution parameter, which is a measure of the ratio of the total computation time of a task over all stages to its deadline. The stage execution time of a task is calculated based on a uniform distribution with mean equal to $\frac{DT}{N}$, where D is the deadline of the task and T is the task resolution. The stage execution times of tasks were allowed to vary up to 10% on either side of the mean. Task preemptions are assumed to be instantaneous, that is, the task switching time is zero. Load is defined as the sum of computation times of all tasks that arrive during the simulation divided by the duration of the experiment. Unless otherwise specified, we use the following default values - system load of 100%, task resolution of 1 : 100, and 5 pipeline stages. The 95% confidence interval for all the utilization values presented in this section is within 0.004 of the mean value, which is not plotted for the sake of legibility.

We first consider the case of aperiodic tasks. Below, we refer to our new process of testing an "equivalent" single-stage system a *meta-schedulability* test. Recall that, in this approach, the entire pipeline is transformed into *one* single-stage system that takes the whole pipeline into account and is subjected to the original end-to-end deadlines. This is in contrast, for example, to approaches that partition end-to-end deadlines into

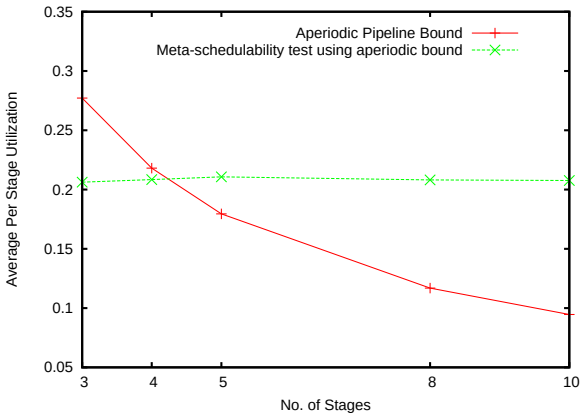

Figure 3.8: Comparison of tests for aperiodic job arrivals

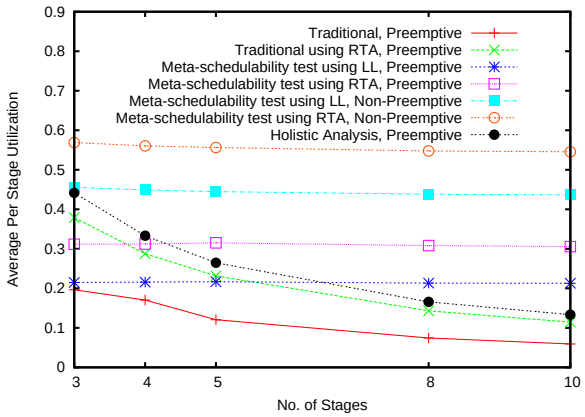

Figure 3.9: Comparison of tests for different pipeline stages

per-stage deadlines then apply uniprocessor analysis to *each* stage *independently*.

For aperiodic tasks, we transform the pipeline into a single stage, and then use in the meta-schedulability test, the uniprocessor aperiodic utilization bound derived in [3]. We compare it with the pipeline bound presented in [34], which is based on the same aperiodic task bound. As there are no previously known techniques to study aperiodic tasks under non-preemptive scheduling, we evaluate the case of aperiodic tasks only under preemptive scheduling. For both the above mentioned tests, while keeping other simulation parameters constant, we varied the number of pipeline stages and measured the utilization, the results of which are shown in Figure 3.8. The average per-stage utilization of the aperiodic pipeline bound presented in [34] decreases linearly with the number of pipeline stages, as it does not account for the overlap in the execution of different pipeline stages. Our meta-schedulability test is able to achieve nearly the same utilization regardless of the number of pipeline stages. For the rest of this section, we shall concern ourselves only with periodic tasks.

We compare our meta-schedulability test with holistic analysis [89], and two implementations of traditional pipeline schedulability tests, which divide the end-to-end deadline into equal individual single stage deadlines. The first implementation, which we call 'traditional', tests for each stage if the sum of the ratios of computation times to per-stage deadlines over all tasks is less than the Liu and Layland bound for periodic tasks. Since this bound is pessimistic when per-stage deadlines are less than periods, our second implementation, which we call 'traditional using RTA', uses response time analysis based on deadline monotonic scheduling to analyze the schedulability of each stage. In this analysis, if the response times on every stage for all tasks are found to be less than their respective per-stage deadlines, then the task set is declared to be schedulable. As explained in the example in Section 3.1, tests that partition end-to-end deadlines to per-stage deadlines (and use single-stage analysis independently on each stage) may be pessimistic because they

assume a worst-case arrival pattern at each stage. Holistic analysis avoids this problem. In holistic analysis, the response time on one stage is considered as the jitter for the next change. The analysis does not divide the end-to-end deadline into single stage deadlines. Nevertheless, by considering the previous stage response time as the jitter, it considers possible that a job is delayed by the same higher priority job on every stage of the pipeline. We compare the above approaches to the performance of our meta-schedulability test under preemptive as well as non-preemptive scheduling. In the following figures, for curves labeled preemptive, the scheduling was preemptive and the preemptive version of the test was used in admission control. Likewise, for curves that are marked non-preemptive, the scheduling was non-preemptive and the non-preemptive version of the test was used. In our meta-schedulability test, we use both the Liu and Layland bound and response time analysis on the resulting single stage system. We did not evaluate the holistic analysis technique applied to non-preemptive scheduling as described in [52], as this adds an extra term to account for blocking due to lower priority jobs and tends to be more pessimistic than holistic analysis applied to preemptive scheduling. The meta-schedulability test applied to non-preemptive scheduling was observed to outperform holistic analysis applied to preemptive scheduling, which in turn, would sustain a higher utilization than holistic analysis applied to non-preemptive scheduling as described in [52]. Likewise, for a similar reason the traditional schedulability analysis was not analyzed under non-preemptive scheduling.

We conducted experiments to measure the average per-stage utilization for different number of pipeline stages, when using admission controllers based on each of the above mentioned tests. In these experiments, task periods were set equal to their *end-to-end* deadlines. Figure 3.9 plots this comparison. Notice that the meta-schedulability test under non-preemptive scheduling using response time analysis as the single stage test, significantly outperforms all other tests. As motivated in Section 3.1, preemption can reduce the overlap in the execution of jobs on different stages, resulting in non-preemptive scheduling performing better than preemptive scheduling in the worst case. We observe that the utilization for both the traditional pipeline tests decrease proportionally with the number of stages in the pipeline system. Holistic analysis outperforms both traditional tests, but its utilization nevertheless decreases with increasing number of pipeline stages. In contrast, our meta-schedulability test sustains nearly the same utilization, regardless of the number of pipeline stages. In other words, the pessimism in declaring task sets schedulable is not dependent on the number of pipeline stages. This property is a result of our delay composition rule. Under preemptive scheduling, the meta-schedulability test outperforms holistic analysis for pipelines longer than 5 stages.

We compared the utilization achieved under preemptive scheduling by our meta-schedulability test based on RTA with holistic analysis, for two different deadline ratio parameters and for different number of pipeline stages. Figure 3.10 plots this comparison. For both analysis techniques, trends similar to those in Figure 3.9

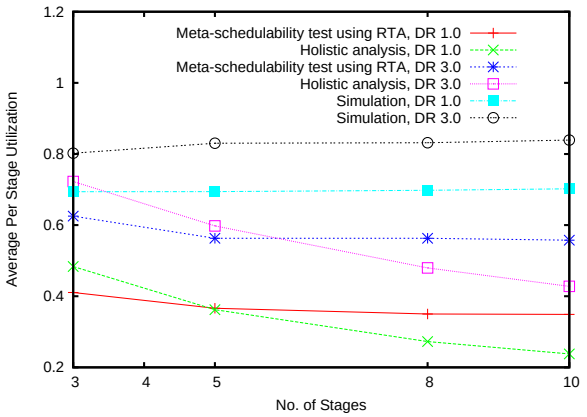

Figure 3.10: Comparison of tests for different stages and different deadline ratio parameter values under preemptive scheduling

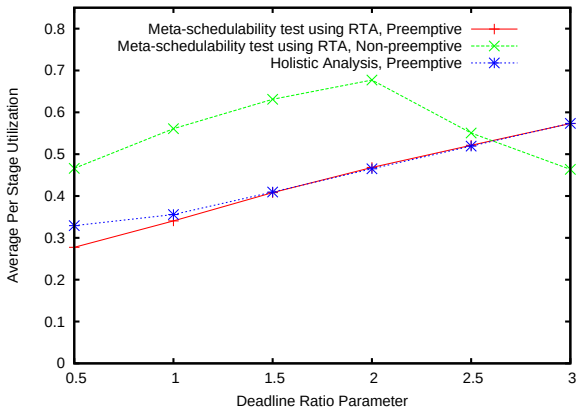

Figure 3.11: Comparison of tests for different deadline ratio parameter values

are observed. It can be observed that as the deadline ratio parameter increases, the achievable per-stage utilization significantly increases. For high deadline ratio parameter values, the deadlines of lower priority tasks are very large compared to those of higher priority tasks (when $DR = 3$, the deadline ratio of the highest to the lowest priority task can be as high as 1000). At most times, some of these lower priority tasks exist in the system and can execute in the background, thereby providing high processor utilization. This figure helps to suggest in some sense, that the worst case situation in terms of reducing the achievable processor utilization, occurs when all tasks have very similar deadlines and stage execution times. Further, the values specified as 'simulation' were the lowest utilization values at which deadline misses were observed in the absence of any admission controller (for the same task parameters). This serves to indicate an upper bound on the achievable utilization.

In order to precisely quantify the space in which non-preemptive scheduling performs better than preemptive scheduling, we compare the performance of the meta-schedulability tests and holistic analysis by varying the deadline ratio parameter DR, while keeping the other parameters equal to their default values. Figure 3.11 plots this comparison for the meta-schedulability test under both preemptive and non-preemptive scheduling, and holistic analysis under preemptive scheduling. Recall that a DR value of x indicates that the end-to-end deadlines of tasks can vary by as much as 10^x. As stage execution times of tasks are chosen proportional to their end-to-end deadline, when the deadlines are very different, the lower priority tasks (with large deadlines) have a large stage execution time. As DR increases, initially, the admitted utilization under preemptive as well as non-preemptive scheduling increases. The reason for this is due to the fact that when lower priority tasks have a larger computation time, they can execute in the background of higher priority tasks leading to better system utilization. However, larger computation times for lower priority tasks imply that higher priority tasks can now be blocked for longer durations under non-preemptive scheduling, which

could lead to missed deadlines and consequently lower utilization sustained by the admission controller. For DR values up to 2, non-preemptive scheduling results in better performance than preemptive scheduling. For DR values greater than 2, the utilization under non-preemptive scheduling decreases, as higher priority jobs are now blocked for longer durations.

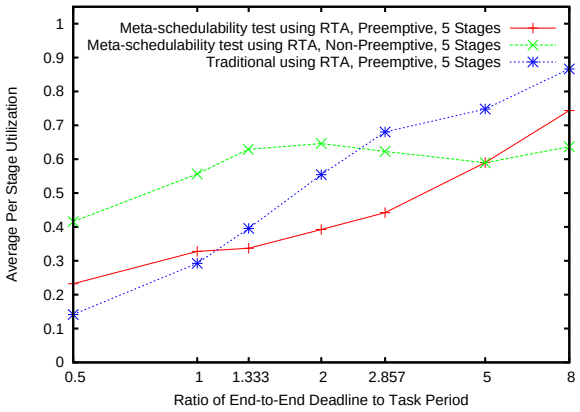

Figure 3.12: Comparison of utilization for different relative values of the end-to-end deadline with respect to the period for 5 pipeline stages

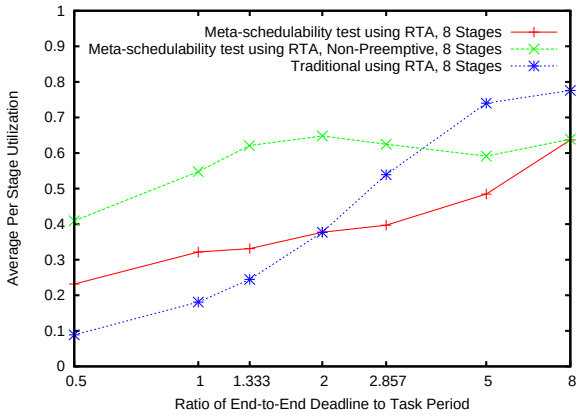

Figure 3.13: Comparison of utilization for different relative values of the end-to-end deadline with respect to the period for 8 pipeline stages

A criticism of the above results is that they favor our tests by setting end-to-end deadlines equal to periods. As mentioned in Section 3.1, traditional tests that partition end-to-end deadlines work very well as long as deadlines are large compared to periods. In order to characterize the break-even point after which our meta-schedulability test under preemptive and non-preemptive scheduling outperforms traditional schedulability analysis under preemptive scheduling, we compared the achievable utilization for different values of the ratio between the end-to-end deadline and the task period, while maintaining the offered system load constant (by proportionately changing the execution times of tasks). Response time analysis was used as the single-stage schedulability test for both the techniques. Figures 3.12 and 3.13 plot this comparison for 5 and 8 pipeline stages, respectively. The x axis is plotted in log scale (base 2). Note that a ratio of 5 for 5 stages, and 8 for 8 stages indicate that the period is equal to the per-stage deadline (for traditional schedulability analysis). When the ratio of the end-to-end deadline to period is higher, the laxity available to jobs is larger, and hence, the utilizations of both techniques are high. Under non-preemptive scheduling, apart from the increased laxity that allows for higher utilizations, there is one other factor that determines the sustainable utilization. As the DR value increases, the deadlines of jobs are larger, and as computation times are chosen proportional to the deadlines, the computation times of jobs also increase. This causes high priority jobs to be blocked for longer durations by lower priority jobs (similar to the trend observed in Figure 3.11), reducing the sustainable utilization. These two opposing forces cause the utilization under non-preemptive scheduling to increase up to a ratio of 2, then decrease until 5, and then increase again. For lower values

of the ratio of the end-to-end deadline to the period, the meta-schedulability test under preemptive and non-preemptive scheduling outperforms the traditional test under preemptive scheduling, while at higher values of the end-to-end deadline, the traditional test performs better. The curve for the traditional test under non-preemptive scheduling would always be lower than the curve under preemptive scheduling, and is not plotted in the figure. The cross-over point, the largest value of the ratio of end-to-end deadline to period where the meta-schedulability test outperforms the traditional test, is larger for non-preemptive scheduling than for preemptive scheduling. Further, the cross-over point is higher for 8 stages than for 5 stages, showing the pessimism of the traditional test with increasing number of pipeline stages.

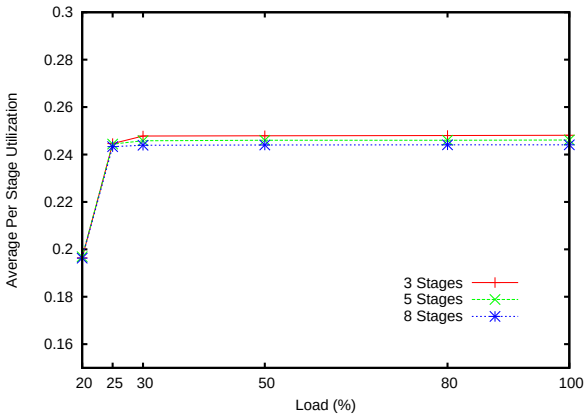

Figure 3.14: Comparison of utilization for different number of pipeline stages and system loads

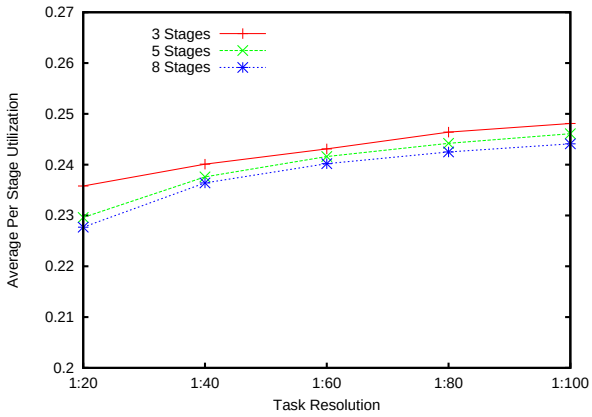

Figure 3.15: Comparison of utilization for different number of pipeline stages and task resolutions

We varied the system load and measured the utilization for our meta-schedulability test under preemptive scheduling together with the Liu and Layland bound for different number of pipeline stages, as shown in Figure 3.14. The loads considered were 20%, 25%, 30%, 50%, 80%, and 100%. The load values represent the load of all tasks presented to the system, and not the load of the admitted tasks. The utilization of the system saturates at a load of about 25%.

We then considered task resolutions of 1:20, 1:40, 1:60, 1:80, and 1:100. For different pipeline stages, we plot the utilization the meta-schedulability test under preemptive scheduling using the Liu and Layland bound in Figure 3.15. Regardless of the number of pipeline stages, the utilization slightly increases with smaller task execution times (with respect to the task deadline), for the same system load. This can be attributed to the fact that the stage additive component is more for larger task execution times, which in turn causes the utilization to be lower.

Chapter 4

Delay Composition for Directed Acyclic Systems

In this chapter, we extend the delay composition results to systems described by arbitrary Directed Acyclic Graphs (DAGs). Informally, the question addressed is as follows: given a distributed task A in a distributed task system of workload W_{dist}, can we systematically construct a uniprocessor task B and a uniprocessor workload W_{uni}, such that if B is schedulable on the uniprocessor, A is schedulable on the distributed system? We show that such a transformation is possible and that it is linear in the number of tasks on A's path. A wide range of existing schedulability analysis techniques can thus be applied to the uniprocessor task set, to analyze the distributed system under both preemptive and non-preemptive scheduling. While the original results apply to priority-based resource scheduling only, we demonstrate how the framework can trivially accommodate resource partitioning (e.g., TDMA) as well.

This chapter is organized as follows. Section 4.1 briefly describes the system model. Section 4.2, generalizes previous pipeline delay composition results to the DAG case, and provides an improved bound for the special case of periodic tasks. We show how partitioned resources can be handled in Section 4.3. In Section 4.4, we present distributed system reduction to a single stage and show how well-known single stage schedulability analysis techniques can be used to analyze acyclic distributed systems. In Section 4.5, we describe a flight control system as an example application, where the theory developed in this chapter can be applied. In Section 4.6, we describe how the system model can be extended to include tasks whose sub-tasks themselves form a DAG. In Section 4.7, we compare the performance of schedulability analysis based on our delay composition theorem with holistic analysis. We describe a quick and dirty way of handling non-acyclic systems, by relaxing cycles within the system in Section 4.8.

4.1 System Model

In this work, we consider a multi-stage distributed system that serves real-time tasks. The system model is similar to the model assumed for pipelined systems. We assume that each task traverses a path of multiple stages of execution and must exit the system within a specified end-to-end deadline. The combination of all such paths forms a DAG. We then extend the above results to tasks whose sub-tasks themselves form a

DAG.

We assume that all stages are scheduled in the same priority order. If some resources are partitioned (e.g., in a TDMA fashion) with priorities applied within partitions, we consider each partition to be a slower prioritized resource and add a delay (the maximum time a task waits for its slot). For example, partitioning communication resources among senders using a TDMA or token-passing protocol is a common approach for ensuring temporal correctness in distributed real-time systems.

4.2 Delay Composition for DAGs

For the purposes of distributed system transformation, let us view the system as seen from the perspective of some job J_1 of relative deadline D_1 whose schedulability is to be analyzed. Job J_1 traverses a multistage path, $Path_1$, in the system, where each stage is a single resource (such as a processor or a network link). While the system may have other resources, we consider only those that J_1 traverses. Let there be N such resource stages, numbered $1, 2, \ldots, N$ in traversal order, constituting $Path_1$. Let the arrival time of any job J_i to stage j of $Path_1$ be denoted $A_{i,j}$. The computation time of J_i at stage j is denoted by $C_{i,j}$, for $1 \leq j \leq N$. If a job J_i does not pass through stage j, then $C_{i,j}$ is zero. Let $C_{i,max}$, denote J_i's largest stage execution time, on stages where both J_i and J_1 execute. Observe that a job J_i ($i \neq 1$) may meet with J_1's path and diverge several times. Let M_i be the number of times the paths of J_i and J_1 meet (for a sequence of one or more consecutive stages that ends with one job using a stage not used by the other). In Sections 4.2.1 and 4.2.2, we derive the proofs for the preemptive and non-preemptive versions of the DAG delay composition theorem, respectively. We then leverage it to present a transformation to an equivalent uniprocessor.

4.2.1 The Preemptive Case

In this section, we bound the maximum delay of J_1 as a function of the execution requirements of higher priority jobs that interfere with it along its path. The pipeline result was proved assuming that all jobs follow the same sequence of stages. However, in the system under consideration, each sub-job J_{i_k} of J_i executes only on a certain consecutive sequence of stages j through j' (say) and does not execute on the other stages. In order to use the pipeline result, we first prove a lemma that generalizes the pipeline delay composition theorem.

For notational simplicity, let us renumber all higher-priority jobs J_{i_k} so they are given a single index increasing in priority order, and let $\bar{Q}$ denote the set of all such jobs including J_1. Further (also for notational simplicity), let us assume that each job has a unique priority. Ties are broken arbitrarily (e.g., in a FIFO

manner).

Lemma 3. *The pipeline delay composition theorem (Equation 3.1) provides a worst-case delay bound for job J_1 in the presence of higher priority jobs (denoted by set $\bar{Q}$ with the inclusion of J_1), each executing on some arbitrary consecutive sequence of stages in the path of J_1.*

$$Delay(J_1) \leq \sum_{J_i \in \bar{Q}} 2C_{i,max} + \sum_{\substack{j \in Path_1, \\ j \leq N-1}} \max_{J_i \in \bar{Q}}(C_{i,j})$$

Proof. The proof is by induction on task priority. While carrying out the induction, we also successively transform each added task, so that it executes on all stages 1 through N with zero execution times on stages on which it did not execute previously. We show that this transformation does not invalidate the delay bound as per the lemma.

The basis step of the lemma is when only J_1 is present in the system. In this case,

$$Delay(J_1) \leq 2C_{1,max} + \sum_{\substack{j \in Path_1, \\ j \leq N-1}} C_{1,j}$$

which is trivially true.

Now, assume that the lemma is true for $k-1$ jobs, $k \geq 2$. We shall prove the lemma when a k^{th} job J_k of highest priority is added. To do so, we need to show that the additional delay due to the presence of J_k is at most $2C_{k,max}$, in addition to J_k's contribution to the stage additive component of the delay (the sum of maximum computation times over all jobs at each stage).

Let J_k execute between stages j and j' in the path of J_1. By adding a zero execution time requirement for J_k on each stage beyond j' in the path of J_1, we do not change the execution intervals or the end-to-end delay of J_1. Now in the system with only $k-1$ jobs, in the absence of J_k, let the delay of J_1 from the time of its arrival till the time it arrives at stage j be x, and the delay from the time it arrives at stage j till the time it completes its execution in the system be y. The end-to-end delay of J_1 is thus $x+y$, when the system has $k-1$ jobs. In the system with job J_k, let the delay of J_1 from the time it arrives at stage j till the time it completes execution on all stages be $y+\Delta$ (Δ is the additional delay caused by J_k).

Consider the system starting from stage j and including all subsequent stages in the path of J_1. All k jobs execute on all the stages (the transformation has been performed for the other $k-1$ jobs), and the system is a pipelined system. We can now apply the pipeline delay composition theorem (Equation 3.1) to this system. From the pipeline delay composition theorem, the worst case delay that J_k can induce J_1, that is the maximum value for Δ, is $2C_{k,max}$, in addition to J_k contributing to the stage additive component of the delay, which is one maximum stage execution time over all jobs for each stage. Thus, Δ is bounded

regardless of the value of x and y and the arrival times of the other jobs. Now, add a zero execution time requirement for J_k on each stage prior to stage j on the path of J_1. As J_k is the highest priority job in the system, as soon as it arrives to the system it would complete its zero execution time requirement on each stage and arrive at stage j instantaneously. Thus, when zero execution time requirements have been added for J_k on stages prior to stage j and beyond stage j', the delay that J_k causes J_1 is still Δ, which is bounded as described above regardless of the arrival times of the other jobs. This proves the induction step and each higher priority job J_k inflicts a delay of at most $2C_{k,max}$ in addition to contributing to the stage additive component.

The lemma is precisely Inequality 4.2 in Section 3.3.1. The same proof applies even for the case of non-preemptive scheduling, except for invoking the non-preemptive pipeline delay composition theorem (Equation 5.6 instead of the preemptive version of the theorem. Thus, under non-preemptive scheduling, Δ is bounded by $C_{k,max}$ (one maximum stage execution time for each higher priority job instead of two), and an additional blocking term determines the delay due to lower priority jobs as shown in Section 4.2.2. □

We now state and prove the delay composition theorem for directed acyclic task graphs.

Preemptive DAG Delay Composition Theorem. *Assuming a preemptive scheduling policy with the same priorities across all stages for each job, the end-to-end delay of a job J_1 of N stages can be composed from the execution parameters of jobs that delay it (denoted by set $\bar{S}$) as follows:*

$$Delay(J_1) \leq \sum_{J_i \in \bar{S}} 2C_{i,max}M_i + \sum_{\substack{j \in Path_1 \\ j \leq N-1}} \max_{J_i \in \bar{S}}(C_{i,j}) \tag{4.1}$$

Proof. The proof of the preemptive DAG delay composition theorem for job J_1 is accomplished by transforming the system to a pipelined system in which the worst case delay of J_1 is no lower than that in the original system. The pipeline delay composition theorem can then be applied to derive a worst case end-to-end delay bound for job J_1.

Consider a job J_i whose path meets with the path of J_1 in the distributed system then splits from it multiple times. Every time the paths of J_i and J_1 meet for one or more consecutive stages, we consider J_i's execution on those stages to be a new job J_{i_k} as shown in Figure 4.1. In other words, we split each J_i into M_i independent jobs, each of which has one or more consecutive common stages of execution with J_1. The transformation effectively relaxes the precedence relations that chain together the jobs J_{i_k} in the original system. The relaxation can only decrease the schedulability of J_1 by making it possible to construct more aggressive worst-case arrival patterns of the higher-priority jobs J_{i_k}. Hence, if J_1 is schedulable in the new

system, it is schedulable in the original system. The new system, however, can be analyzed by the pipeline result in Equation 3.1 and Lemma 3.

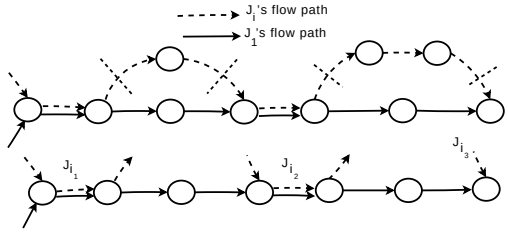

Figure 4.1: Figure illustrating splitting job J_i into M_i independent sub-jobs.

Let set $\bar{Q}$ denote the set of all higher priority jobs J_{i_k} over all jobs J_i and including J_1. We can now apply the pipeline delay composition theorem to bound the worst case end-to-end delay of J_1. We have:

$$Delay(J_1) \leq \sum_{J_i \in \bar{Q}} 2C_{i,max} + \sum_{\substack{j \in Path_1, \\ j \leq N-1}} \max_{J_i \in \bar{Q}}(C_{i,j}) \tag{4.2}$$

Since each job J_i gave rise to M_i sub-jobs J_{i_k}, the summation over all jobs J_{i_k} in set $\bar{Q}$ (in the first term of the bound above) can be rewritten as a double summation over jobs J_i in $\bar{S}$ and their M_i sub-jobs. Similarly, the maximization in the second term can also be broken into two as follows:

$$Delay(J_1) \leq \sum_{J_i \in \bar{S}} \sum_{k=1}^{M_i} 2C_{i_k,max} + \sum_{\substack{j \in Path_1, \\ j \leq N-1}} \max_{J_i \in \bar{S}} (\max_{k \leq M_i} (C_{i_k,j}))$$

This is equivalent to:

$$Delay(J_1) \leq \sum_{J_i \in \bar{S}} 2C_{i,max} M_i + \sum_{\substack{j \in Path_1, \\ j \leq N-1}} \max_{J_i \in \bar{S}}(C_{i,j}) \tag{4.3}$$

This proves the preemptive DAG delay composition theorem. □

The above theorem presents a delay bound for J_1 given any arbitrary set of higher priority jobs $\bar{S}$. For the special case where the higher priority jobs are invocations of periodic tasks, denoted by set R, an improved delay bound can be derived based on the observation that not all sub-jobs of each invocation of task $T_i \in R$ contribute to the delay of J_1. Let x_i denote the number of invocations of task T_i that can potentially contribute to the delay of J_1 (the number of invocations of T_i that belong to set $\bar{S}$). The following corollary derives this improved bound for periodic tasks.

Corollary 1. *Under preemptive scheduling, the end-to-end worst-case delay bound for a job J_1 of a lowest priority task T_1, in the presence of higher priority periodic tasks (denoted by set R) is given by:*

$$Delay(J_1) \leq \sum_{T_i \in R} 2C_{i,max}(x_i + M_i) + \sum_{\substack{j \in Path_1 \\ j \leq N-1}} \max_{T_i \in R}(C_{i,j}) \tag{4.4}$$

Proof. Each invocation of T_i has M_i sub-jobs, and there are x_i such invocations in set $\bar{S}$. The key observation

is that not all $x_i \times M_i$ sub-jobs of T_i can delay J_1, and by removing the sub-jobs that cannot delay J_1 from set $\bar{Q}$, an improved delay bound can be obtained for periodic tasks. To see that, consider the delay of one invocation J_1 of the periodic task under consideration. This invocation makes forward progress along its path and never revisits a stage. Hence, for example, if all M_i sub-jobs of one invocation of T_i delay J_1, it implies that J_1 has already progressed past a certain stage on its path (specifically, past the last stage, say g, where the paths of T_i and T_1 meet). Therefore, sub-jobs of future invocations of T_i that may execute later at those already traversed stages (i.e., stages prior to g) will not interfere with J_1. Extending this argument, if $y_1 \leq M_i$ sub-jobs of the first invocation of T_i delay J_1, then only $y_2 \leq M_i - y_1 + 1$ sub-jobs of the second invocation can delay J_1. Likewise, only $y_3 \leq M_i - (y_1 + y_2) + 2$ sub-jobs of the third invocation can delay J_1. Therefore, the total number of sub-jobs of T_i that delay J_1 is bounded by $y_1 + y_2 + \ldots + y_{x_i} \leq x_i + M_i$. Thus, to calculate the worst-case delay for J_1, we can discard all but $x_i + M_i$ sub-jobs of T_i from set $\bar{Q}$. This new system, however, can be analyzed by the pipeline result as before. The corollary follows by grouping all sub-jobs belonging to the same periodic task together.

$$Delay(J_1) \leq \sum_{J_i \in \bar{Q}} 2C_{i,max} + \sum_{\substack{j \in Path_1, \\ j \leq N-1}} \max_{J_i \in \bar{Q}}(C_{i,j}) \leq \sum_{T_i \in R} 2C_{i,max}(x_i + M_i) + \sum_{\substack{j \in Path_1 \\ j \leq N-1}} \max_{T_i \in R}(C_{i,j})$$

□

4.2.2 The Non-Preemptive Case

Next, we bound the maximum delay of J_1 under non-preemptive scheduling. Unlike the previous case, here J_1 might also be delayed by lower-priority jobs, collectively denoted by set $\underline{S}$. In particular, it may be delayed by up to one such job on each stage. The following theorem states the new delay bound.

Non-preemptive DAG Delay Composition Theorem. *Assuming a non-preemptive scheduling policy with the same priorities across all stages for each job, the end-to-end delay of a job J_1 of N stages can be composed from the execution parameters of other jobs that delay it (denoted by set S) as follows:*

$$Delay(J_1) \leq \sum_{J_i \in \bar{S}} C_{i,max} M_i + \sum_{\substack{j \in Path_1 \\ j \leq N-1}} \max_{J_i \in S}(C_{i,j}) + \sum_{j \in Path_1} \max_{J_i \in \underline{S}}(C_{i,j}) \tag{4.5}$$

Proof. To bound the worst case delay for a job J_1 under non-preemptive scheduling, we first transform the task set by removing all lower-priority jobs, and instead adding to the computation time of J_1 on each stage i the maximum blocking delay due to jobs in $\underline{S}$. Let us call the adjusted computation time, $C_{1',j}$. Hence, $C_{1',j} = C_{1,j} + \max_{J_i \in \underline{S}}(C_{i,j})$. This results in a system of only J_1 and higher-priority jobs. Observe that if the new system is schedulable so is the original one because we extended J_1's computation time by the worst case amount of time it could have been blocked by lower priority jobs. We then cut each higher-priority job

J_i into M_i sub-jobs as we did in the preemptive case, and let $\bar{Q}$ denote the set of all such sub-jobs including J_1. The resulting system is a task pipeline to which the non-preemptive pipeline delay composition theorem (Equation 3.2) applies. According to this theorem:

$$\begin{aligned}
Delay(J_1) &\leq \sum_{J_i \in \bar{Q}} C_{i,max} + \sum_{\substack{j \in Path_1, \\ j \leq N-1}} \max_{J_i \in \bar{Q}}(C_{i,j}) \\
&\leq \sum_{J_i \in \bar{S}} \sum_{k=1}^{M_i} C_{i_k,max} + \sum_{\substack{j \in Path_1, \\ j \leq N-1}} \max(\max_{\substack{J_i \in \bar{S}, \\ k \leq M_i}}(C_{i_k,j}), C_{1',j}) \\
&\leq \sum_{J_i \in \bar{S}} C_{i,max} M_i + \sum_{\substack{j \in Path_1, \\ j \leq N-1}} \max(\max_{J_i \in \bar{S}} C_{i,j}, C_{1',j}) \\
&\leq \sum_{J_i \in \bar{S}} C_{i,max} M_i + \sum_{\substack{j \in Path_1, \\ j \leq N-1}} \max_{J_i \in \bar{S}}(C_{i,j}) + \sum_{j \in Path_1} \max_{J_i \in \underline{S}}(C_{i,j}) \qquad (4.6)
\end{aligned}$$

Inequality 4.6 follows by replacing $C_{1',j}$ by $C_{1,j} + \max_{J_i \in \underline{S}}(C_{i,j})$ and making the delay due to lower priority jobs a separate term. This proves the non-preemptive DAG delay composition theorem. □

For the special case of periodic tasks, an improved bound can be derived as before. Let the set of all periodic tasks be denoted by R. Let $\bar{R}$ denote the set of higher priority tasks including T_1 and $\underline{R}$ denote the set of lower priority tasks.

Corollary 2. *Under non-preemptive scheduling, the end-to-end delay bound for a job J_1 of task T_1, in the presence of other periodic tasks (denoted by set R) is given by:*

$$Delay(J_1) \leq \sum_{T_i \in \bar{R}} C_{i,max}(x_i + M_i) + \sum_{\substack{j \in Path_1, \\ j \leq N-1}} \max_{T_i \in \bar{R}}(C_{i,j}) + \sum_{j \in Path_1} \max_{T_i \in \underline{R}}(C_{i,j}) \qquad (4.7)$$

Proof. The proof is similar to the preemptive case, and we do not repeat the proof in the interest of brevity. □

4.3 Handling Partitioned Resources

The delay composition theorem described so far, is only applicable to systems where resources are scheduled in priority order. However, resources such as network bandwidth are often *partitioned* among jobs, for example, using a TDMA protocol. In such a partitioned resource, a job may access the resource only during its reserved time-slot. Multiple jobs can share a time-slot and be scheduled in priority order within it.

Consider a stage j that is a partitioned resource. Let job J_i be allocated a slice that is served for B_{slice} time units every B_{total} time units. As shown in Figure 4.2, this is no worse than having a dedicated resource

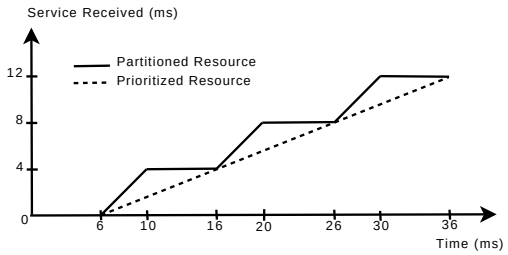

Figure 4.2: Illustration of conversion of a partitioned resource into a prioritized resource.

that is slower by a factor B_{slice}/B_{total} and that introduces an access delay of at most $B_{total} - B_{slice}$.

Figure 4.2 illustrates the service received by a set of tasks over time for the original partitioned resource and for its corresponding dedicated prioritized resource, when $B_{slice} = 4ms$ and $B_{total} = 10ms$. Note that the service received under the prioritized resource will always be less than in the partitioned resource, causing tasks to be delayed longer. Hence, this transformation is safe in that if the tasks in the transformed system are schedulable, so are the tasks in the original system.

When analyzing the end-to-end delay of J_1, the computation time of J_1 on the new prioritized resource j can be taken as $C_{1,j} \times \frac{B_{total}}{B_{slice}} + (B_{total} - B_{slice})$ (the additional delay is subsumed in the computation time). The computation time of all other jobs in the same slice would be $C_{i,j} \times \frac{B_{total}}{B_{slice}}$.

Once this transformation is conducted for all partitioned resources that J_1 encounters in the system, the delay composition theorem can be directly applied to compute the worst case end-to-end delay of J_1.

4.4 Transforming Distributed Systems

The preemptive and non-preemptive DAG delay composition theorems derived in Section 4.2, can be used to reduce a given distributed acyclic system to an equivalent single stage system, similar to the reduction performed for pipeline systems in Chapter 3. Let S_{wc} denote the worst-case set of jobs that can potentially delay J_1.

In Sections 4.4.1 and 4.4.2, we briefly show how an equivalent uniprocessor system can be created to analyze schedulability of the original system under preemptive and non-preemptive scheduling, respectively. When the system consists of partitioned resources, we assume that the transformation described in Section 4.3 has already been performed.

4.4.1 Preemptive Scheduling Transformation

The form of the DAG delay composition theorem suggests a reduction to a uniprocessor system in which the lowest-priority uniprocessor job suffers the delay stated by the theorem. This reduction to a single stage system is conducted by (i) replacing each higher priority job J_i in $\bar{S}_{wc}$ by a single stage job J_i^* of execution time equal to $2C_{i,max}M_i$, and (ii) replacing J_1 with a lowest-priority job, J_1^* of execution time

equal to $2C_{1,max} + \sum_{j \in Path_1, j \leq N-1} \max_i(C_{i,j})$ (the second term is the stage-additive component), and deadline same as that of J_1. The delay of J_1^* on the hypothetical uniprocessor adds up to the delay bound as expressed in the right hand side of Inequality 4.1. By the delay composition theorem, the total delay incurred by J_1 in the acyclic distributed system is no larger than the delay of J_1^* on the uniprocessor. Thus, if J_1^* completes prior to its deadline in the uniprocessor, so will J_1 in the acyclic distributed system.

4.4.2 Non-Preemptive Scheduling Transformation

Under non-preemptive scheduling, we reduce the DAG into an equivalent single stage system that runs *preemptive* scheduling as before. This is achieved by (i) replacing each job J_i in $\bar{S}_{wc}$ by a single stage job J_i^* of execution time equal to $C_{i,max}M_i$, and (ii) replacing J_1 by a lowest-priority job, J_1^* of execution time equal to $C_{1,max} + \sum_{j \in Path_1, j \leq N-1} \max_{J_i \in \bar{S}_{wc}}(C_{i,j}) + \sum_{j \in Path_1} \max_{J_i \in \underline{S}}(C_{i,j})$ (which are the last two terms in Inequality (4.5)), and deadline same as that of J_1. Note that the execution time of J_1^* includes the delay due to all lower priority tasks. Further, in the above reduction, the hypothetical single stage system constructed is scheduled using preemptive scheduling, while the original DAG was scheduled using non-preemptive scheduling. This is because we only care to match the sum of the delay experiences by J_1 and J_1^* in their respective systems. By the delay composition theorem, the total delay incurred by J_1 in the acyclic distributed system under non-preemptive scheduling is no larger than the delay of J_1^* on the uniprocessor under preemptive scheduling, since the latter adds up to the delay bound expressed on the right hand of Inequality (4.5).

4.5 A Flight Control System Example

In this section, we describe a practical problem faced in flight control systems and explicate how the theory developed in this work can efficiently solve the problem. In order to keep the example simple and illustrative, we have modified certain attributes of the system. We also show how network scheduling (as a partitioned resource) can easily be handled within the assumed system model. The purpose of the example is to illustrate how the theory developed in this chapter can be applied, and is not intended as a comparison with existing theory on schedulability analysis for distributed systems. Such a comparison is presented in the evaluation section.

A flight control system (with some sub-systems excluded for simplicity) is shown in Figure 4.3(a). The Flight Guidance System (FGS) receives periodic sensor readings from the Attitude and Heading Reference System (AHRS) and the Navigation Radio (NAV_RADIO). The sensory information gets processed by the FGS and the Auto-Pilot (AP), and the elevator servo component performs the actuation. The Flight Control

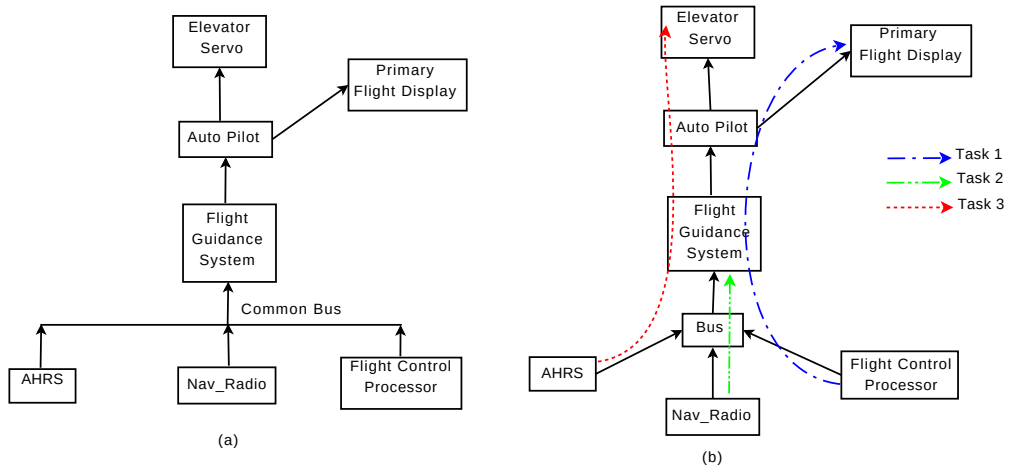

Figure 4.3: (a) Example flight control system (b) The different flows in the system, with the bus abstracted as a separate stage of execution

Processor (FCP) is responsible for input commands from the pilot and display settings. Commands from the FCP need to be processed by the FGS, and display information needs to be transmitted to the Primary Flight Display (PFD). The actual flight control system uses dedicated buses to carry information from one unit to another. However, in order to illustrate how network scheduling can be handled, we assume the presence of a common bus connecting the FGS to the various units that feed into it. Further, we assume a simple TDMA protocol for bus access, which is a common approach to temporal isolation in avionics.

The various tasks that constitute the system are shown in Figure 4.3(b). Task T_3, the highest priority task, carries periodic sensory information from AHRS to the FGS. The FGS then processes this information, the AP generates commands, and the Servo performs the actuation (adjusts the pitch). Task T_2 carries sensor readings from the NAV_RADIO to the FGS periodically. Commands from the FCP are routed to the PFD through the FGS and AP in task T_1 and is the lowest priority task in the system. T_3 belongs to a separate class on the bus, and T_2 and T_1 belong to a single class. The TDMA protocol on the bus employs a period of 10ms, and allots the first 4ms to the AHRS, the next 6ms to the NAV_RADIO (T_2) and FCP (T_1). Scheduling of tasks at each stage is preemptive and prioritized. Worst case computation times (hypothetical) for the tasks at different stages, their periods and deadlines, are shown in Table 4.1 (all values in milli-seconds). A hyphen denotes that the task does not execute on the corresponding stage. The value shown for the tasks under 'Bus' denotes the time taken to carry the periodic information on the bus to the FGS.

For brevity, we analyze schedulability of T_1 only. Schedulability of other tasks can be analyzed similarly. We first need to transform the partitioned bus, into a prioritized resource as described in Section 4.3. T_1 and T_2 together have a time slot of 6ms every 10ms. The partitioned bus is no worse than a dedicated prioritized resource providing service to T_1 and T_2 at a rate slower by a fraction $\frac{6}{10}$, and causing an additional delay of

	T_1	T_2	T_3
AHRS	-	-	10
NAV	-	10	-
FCP	15	-	-
FGS	10	20	15
AP	15	-	20
Servo	-	-	10
PFD	10	-	-
Period	500	250	100
Deadline	450	200	100
Bus	15	6	4

Table 4.1: Task characteristics (in ms)

$10-6=4ms$. The computation time of T_1 on the transformed bus can be taken as $15\times\frac{10}{6}+(10-6)=29ms$. The computation time of T_2 on the bus is $6\times\frac{10}{6}=10ms$. From the computation times provided in Table 4.1, we can obtain $C_{3,max}=C_{2,max}=20ms$ and $C_{1,max}=29ms$ (on the bus); $SM_{3,1}=SM_{2,1}=SM_{1,1}=0$.

As shown in Section 4.4.1, the reduction for this system scheduled preemptively can be conducted by (i) replacing T_3 and T_2 by equivalent single stage tasks T_3^* and T_2^*, with execution times $C_3^*=2C_{3,max}=40ms$ and $C_2^*=2C_{2,max}=40ms$, and periods $P_3^*=100ms$ and $P_2^*=250ms$; (ii) adding a lowest priority task T_e^* with computation time $C_e^*=C_{3,max}+C_{2,max}+C_{1,max}+\sum_{j=FCP,Bus,FGS,AP}\max_i(C_{i,j})$, i.e., $C_e^*=20+20+29+15+29+20+20=153ms$ and having a deadline of 450ms. Applying the response time analysis test [8], we obtain the worst case delay of T_e^* in the single stage system as $393ms$, which is less than the deadline. As T_e^* is schedulable on the hypothetical uniprocessor system, from the delay composition theorem, T_1 is schedulable in the flight control system.

An important requirement in such time-critical systems is to have complete knowledge of dependencies and to be able to determine how changes in the timing properties of one task would affect the schedulability of the system. This is especially true for a flight control system, given its complexity in the number of interacting components (the example provided here is a much simplified version of the actual problem). The analysis developed in this work can be applied on the fly to test schedulability, when the timing properties of individual tasks change during the design and development of the system. For example, consider the case where changes in packet format or size causes an increase in the computation time of T_3 at the FGS and AP to $20ms$ and $25ms$, respectively. Schedulability analysis can be easily performed as before, to test if the system is still schedulable. For brevity, we only show the schedulability of T_1. We obtain $C_3^*=50ms$, $C_2^*=40ms$, and $C_e^*=205ms$. Analyzing the new hypothetical single stage system, we obtain the worst case delay for T_e^* as $485ms$. As T_e^* is still schedulable, T_1 is guaranteed to complete before its end-to-end deadline in the distributed system.

4.6 Handling Tasks whose Sub-Tasks Form a DAG

In the discussion so far, we have only considered tasks whose sub-tasks form a *path* in the Directed Acyclic Graph. In this section, we describe how this can be extended to tasks whose sub-tasks themselves form a DAG. We shall refer to such tasks as DAG-tasks. Figure 4.4(a) shows an example task, whose sub-tasks form a DAG. Edges in the DAG, as before, indicate precedence constraints between sub-tasks and each sub-task executes on a different resource. A sub-task s can execute only after all sub-tasks which have edges to sub-task s have completed execution. In the task shown in the figure, sub-task 5 can execute only after sub-tasks 2 and 3 have completed execution. We call this a 'merger' of sub-tasks. Note that a split, that is, edges from one sub-task s to two or more sub-tasks indicate that once sub-task s completes, it spawns multiple sub-tasks each executing in parallel. It can be observed from the example in Figure 4.4(a), that once sub-task 1 completes, it spawns sub-tasks 2 and 3 that can execute in parallel on different stages.

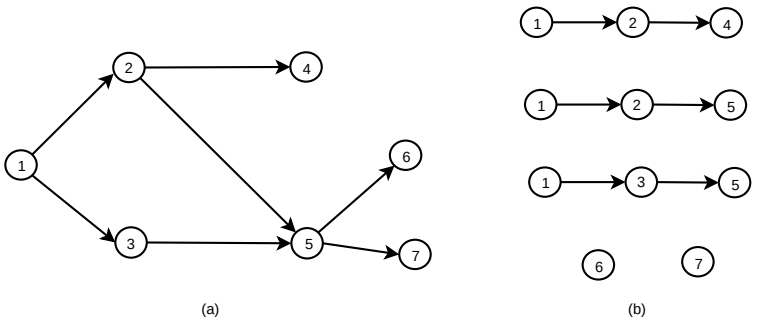

Figure 4.4: (a) Figure showing an example of a DAG-task (b) Different parts of the DAG-task that need to be separately analyzed to analyze schedulability of the DAG-task.

As the delay composition theorem only addresses tasks which execute in sequential stages (that is, the sub-tasks form a path in the DAG) and does not consider DAG-tasks, we need to break the DAG-task into smaller tasks which form a path of the DAG. This is carried forth as follows. Similar to traditional distributed system scheduling, artificial deadlines are introduced after each merger of sub-tasks. Each split in the DAG creates additional paths that need to be analyzed (the number of additional paths is one less than the fan-out). In the example DAG-task, an artificial deadline is imposed after sub-task 5. Sub-tasks 6 and 7 are analyzed independently using any single stage schedulability test. As there are two splits within sub-tasks 1 through 5, there are 3 paths that need to be analyzed as shown in Figure 4.4(b). The path 1-2-4 is analyzed independently using the meta-schedulability test and this sequence of sub-tasks need to complete within the end-to-end deadline of the DAG-task. The paths 1-2-5 and 1-3-5 can be independently analyzed using the meta-schedulability test, with their deadline set as the artificial deadline. Sub-tasks 6 and 7 need to complete in a duration at most equal to the end-to-end deadline of the DAG-task minus the artificial deadline set for sub-task 5. If all the parts of the DAG-task are determined to be schedulable, then

the DAG task is deemed to be schedulable.

As observed in [38], imposing artificial deadlines add to the pessimism of the schedulability analysis. The use of the delay composition theorem reduces the need to impose artificial deadlines to only stages in the execution where two or more sub-tasks merge. This is in contrast to traditional distributed schedulability analysis, that imposes artificial deadlines after each stage of execution, causing the pessimism to quickly increase with system scale.

4.7 Simulation Results

In this section, we evaluate the preemptive and non-preemptive schedulability analysis techniques based on our DAG delay composition theorems. We enhance our custom-built simulator to model a distributed system with directed acyclic flows. We consider only periodic tasks, and further assume that partitioned resources within the system have been transformed into resources scheduled in priority order as described in Section 4.3, and focus this evaluation on prioritized resources. An admission controller based on our reduction of the multistage distributed system to a single stage is built.

Although the meta schedulability test derived in this work is valid for any fixed priority scheduling algorithm, we only present results for deadline monotonic scheduling due to its widespread use. Each point in the figures below represents average utilization values obtained from 100 executions of the simulator, with each execution running for 80000 task invocations. When comparing different admission controllers, each admission controller was allowed to execute on the same 100 task sets.

The default number of nodes in the distributed system is assumed to be 8. Each task on arrival requests processing on a sequence of nodes (we do not consider DAG tasks in this evaluation), with each node in the distributed system having a probability of NP (for Node Probability) of being selected as part of the route. The task's route is simply the sequence of selected nodes in increasing order of their node identifier. The default value of NP is chosen as 0.8. Other simulation parameters are chosen similar to the parameters in Section 3.7. The default value for DR is 0.5. We used a task resolution of 1/100. The 95% confidence interval for all the utilization values presented in this section is within 0.02 of the mean value, which is not plotted for the sake of legibility.

We first study the achievable utilization of our meta-schedulability test using both the Liu and Layland bound and response time analysis, for both preemptive as well as non-preemptive scheduling. We compare this with holistic analysis [89], applied to preemptive scheduling, for different number of nodes in the DAG, the results of which are shown in Figure 4.5. While extensions to holistic analysis have been proposed (such as [69]), we use holistic analysis as a comparison as these extensions are targeted to handle offsets

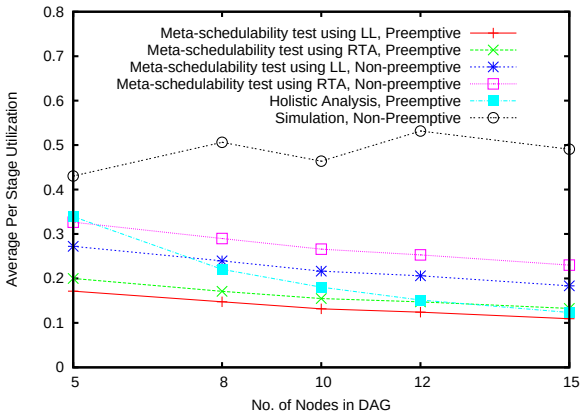

Figure 4.5: Meta-schedulability test vs. holistic analysis for different number of nodes in DAG

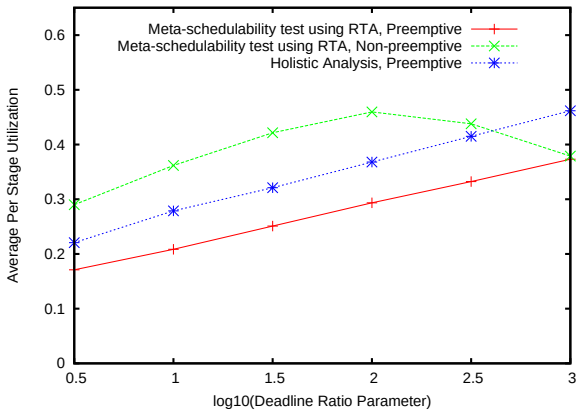

Figure 4.6: Meta-schedulability test vs. holistic analysis for different deadline ratio parameters

(we do not consider offsets in our analysis). Further, they suffer from similar drawbacks as holistic analysis such as poor scalability and requiring global knowledge of all tasks in the system. For meta-schedulability test curves that are marked preemptive, the scheduling was preemptive and the preemptive version of the test was used in admission control. Likewise, for the meta-schedulability test curves that are marked non-preemptive, the scheduling was non-preemptive and the non-preemptive version of the test was used. We only evaluated holistic analysis applied to preemptive scheduling as presented in [89], as the non-preemptive version presented in [52] adds an extra term to account for blocking due to lower priority tasks and tends to be more pessimistic than the preemptive version, and the corresponding curve would always be lower than the curve for preemptive scheduling.

It can be observed from Figure 4.5, that even for an eight node DAG, non-preemptive scheduling analyzed using our meta-schedulability test significantly outperforms preemptive scheduling analyzed using both holistic analysis and our meta-schedulability test. As the utilization curve for holistic analysis applied to non-preemptive scheduling would be lower than the curve for the preemptive scheduling version of holistic analysis, non-preemptive scheduling analyzed using our meta-schedulability test would also outperform the non-preemptive version of holistic analysis. A drawback of holistic analysis is that it analyzes each stage separately assuming the response times of tasks on the previous stage to be the jitter for the next stage. It therefore assumes that every higher priority job will delay the lower priority job at every stage of its execution, ignoring possible pipelining between the executions of the higher and lower priority jobs. This causes holistic analysis to become increasingly pessimistic with system size when periods are of the order of end-to-end deadlines (as opposed to per-stage deadlines). As motivated in [40], preemption can reduce the overlap in the execution of jobs on different stages, resulting in non-preemptive scheduling performing better than preemptive scheduling in the worst case.

In order to estimate when deadlines are actually being missed, and to evaluate the pessimism of the admission controllers, we conducted simulations to identify the lowest utilization at which deadlines are missed. The curve labeled 'Simulation' in Figure 4.5 presents the results from simulations of the lowest utilization at which deadline misses were observed for different number of nodes in the system when non-preemptive scheduling was employed. The corresponding curve for preemptive scheduling, was within 0.02 of those of non-preemptive scheduling, and we don't show the values here for the sake of clarity (the reader must bear in mind that task sets were generated randomly, and that the task sets do not represent worst case scenarios). Each point for the simulation curve was obtained from 500 executions of the simulator in the absence of any admission controller, with each execution considering a workload with utilization close to where deadline misses were being observed. We observe that the meta-schedulability test curves degrade only marginally with increasing scale, while the performance of holistic analysis degrades more rapidly.

To precisely evaluate the scenarios under which non-preemptive scheduling performs better than preemptive scheduling in distributed systems, we conducted experiments varying the deadline ratio parameter (DR) while keeping the other parameters equal to their default values. Figure 4.6 plots a comparison of the meta-schedulability test under both preemptive as well as non-preemptive scheduling, with holistic analysis for different DR values ranging between 0.5 and 3.0. A DR value of x indicates that the end-to-end deadlines of tasks can differ by as much as 10^x. As stage execution times are chosen proportional to the end-to-end deadline, when the end-to-end deadlines of tasks are widely different, the lower priority tasks (those with large deadlines) have a large stage execution time. Initially, as DR increases, the utilization for both preemptive as well as non-preemptive scheduling increases, as lower priority tasks can execute in the background of higher priority tasks resulting in better system utilization. Up to $DR = 2$, non-preemptive scheduling (together with the non-preemptive version of the meta-schedulability test) results in better performance than preemptive scheduling (together with the preemptive version of the test). However, for values of DR greater than 2, that is, the end-to-end deadlines vary by over two orders of magnitude, preemptive scheduling performs better than non-preemptive scheduling. The achievable utilization under non-preemptive scheduling decrease beyond a DR value of 2, as higher priority tasks can now be blocked for a longer duration under non-preemptive scheduling, leading to a greater likelihood of deadline misses.

We conducted a similar comparison of the three admission controllers as in the previous experiment, but for different values of the Node Probability (NP) parameter, which is the probability with which each node in the system is chosen as part of the route of each task. This comparison is shown in Figure 4.7, for different NP parameter values ranging between 0.2 to 1.0 in steps of 0.2. Note that the NP parameter of 1.0 denotes a perfectly pipelined system, where each task executes sequentially on all the nodes in the

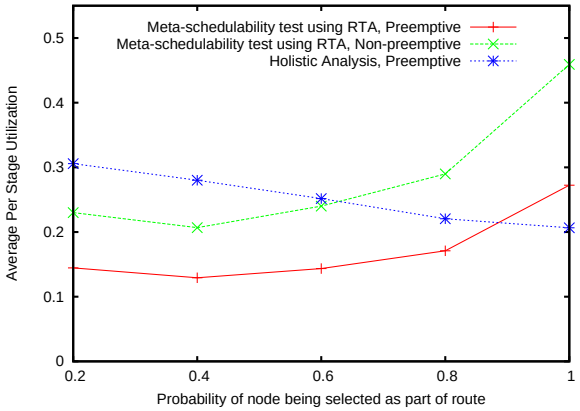

Figure 4.7: Comparison of meta-schedulability test using both preemptive and non-preemptive scheduling with holistic analysis for different route probabilities

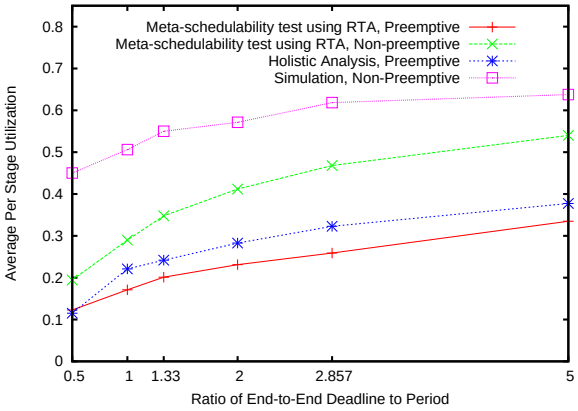

Figure 4.8: Meta-schedulability test vs. holistic analysis for different ratios of end-to-end deadline to task periods

distributed system. For small values of NP, the number of stages on which each task executes is low, and as observed in Figure 4.5, holistic analysis performs better than the meta-schedulability test. However, for larger values of NP, each task traverses more stages in the distributed system, causing holistic analysis to become more pessimistic in its worst case delay bound. The meta-schedulability test using non-preemptive scheduling performs the best for NP values greater than 0.6.

The above results have all been obtained by setting the end-to-end deadlines equal to the periods of tasks. Figure 4.8 plots a comparison of the meta-schedulability test under preemptive and non-preemptive scheduling with holistic analysis for different ratios of the end-to-end deadlines to the periods. When the ratio of the end-to-end deadline to period is higher, the laxity available to jobs is larger, and hence, the utilization of all the three analysis techniques are high. The meta-schedulability test under non-preemptive scheduling consistently outperforms preemptive scheduling analyzed using either the meta-schedulability test or holistic analysis. As holistic analysis applied to non-preemptive scheduling (curve not shown) would perform worse than the preemptive scheduling version of holistic analysis, it would also perform worse than the meta-schedulability test applied to non-preemptive scheduling. Similar to Figure 4.5, the curve labeled as 'simulation' plots the lowest utilization at which deadline misses were observed obtained from simulations under non-preemptive scheduling in the absence of any admission controller. The corresponding values for preemptive scheduling were close to those obtained for non-preemptive scheduling and are not presented here for the sake of clarity. We observe that our analysis tends to be less pessimistic for larger values of the ratio between the end-to-end deadline and the period.

4.8 Handling Non-Acyclic Systems

A key step in deriving the DAG delay composition theorems was to split each higher priority job J_i into M_i sub-jobs J_{i_k}, each executing on one or more consecutive common stages with J_1. The precedence constraints in the arrival times of the different sub-jobs can be relaxed by assuming that each sub-job arrives independently of the others. This independence assumption can only result in a more pessimistic delay analysis for J_1. The same transformation of splitting jobs into sub-jobs and assuming independent arrivals for sub-jobs, can also be conducted for non-acyclic higher priority jobs (jobs that visit a stage more than once). Each visit to a stage can be considered as an independent arrival of a sub-job. In the case where J_1 (the job under consideration) itself has loops in its path, then J_1 can be split into sub-jobs each of which is acyclic. The DAG delay composition theorem can then be used to determine the worst-case delay for each sub-job, and the worst-case delay for J_1 can be estimated as the sum of the worst-case delays of each of its sub-jobs. Even with only one loop in the task path, there may be multiple ways in which the loop can be broken. For example, suppose that a task traverses stages 1, 2, 3, and then revisits stage 1. This loop 1-2-3-1 can be broken as either (1, 2-3-1) (one sub-job on stage 1 and another that executes along the path 2-3-1), (1-2, 3-1), or (1-2-3, 1). This choice becomes an art of design, and the choice that maximizes pipelining and minimizes the number of independent sub-jobs would typically yield the best delay bound. A better characterization of the precedence constraints between sub-jobs (instead of assuming them to be independent) could yield a more accurate delay bound for non-acyclic task systems, which we describe in the next chapter.

Chapter 5

End-to-End Delay Analysis of Arbitrary Distributed Systems

In this chapter, we significantly extend the scope of applicability of our delay composition results by introducing the first reduction-based schedulability analysis technique that applies to distributed systems with *non-acyclic* task graphs. Informally, a task graph is non-acyclic if task flows in the underlying distributed system include cycles. Most common types of traffic do, in fact, have non-acyclic behavior. For example, request-response traffic in client-server systems includes flows (of requests) from client machines to server machines and flows (of responses) in the reverse direction. Hence, analysis of end-to-end latency entails analysis of a non-acyclic task flow. Reliability mechanisms that transmit and process acknowledgments, as well as token passing mechanisms are other examples of systems with non-acyclic task flows.

The fundamental problem in handling task graphs that contain cycles is that the arrival pattern of jobs to a particular node in the system is directly or indirectly dependent on the rate at which jobs exit the node downstream, but that downstream pattern is in turn dependent on the load of the node under consideration and hence on this node's arrival pattern. This is a cyclic dependency. As described in Chapter 2, existing schedulability analysis techniques become too pessimistic or complicated for non-acyclic task systems.

Being a reduction-based approach to schedulability analysis [42], the derived end-to-end delay bound provides a means by which the schedulability analysis of tasks in a distributed system with cycles can be reduced to that of analyzing an equivalent hypothetical uniprocessor. Thus, well-known uniprocessor analysis techniques can be used to analyze the schedulability of tasks in arbitrary distributed systems.

The rest of this chapter is organized as follows. We briefly describe the system model in Section 5.1 and state and prove the end-to-end delay bound for jobs in non-acyclic systems in Section 5.2. In Section 5.3, we briefly describe how the end-to-end delay bound can be used to reduce the schedulability problem of tasks in distributed systems to that of analyzing an equivalent hypothetical uniprocessor. We illustrate the advantage of using the analysis technique presented in this chapter using an example in Section 5.4 and through simulation studies in Section 5.5.

5.1 System Model

Our model of non-acyclic distributed processing consists of a distributed system of N nodes and a set of real-time periodic or aperiodic jobs. Each node is a resource, which is anything that is allocated to jobs in priority order. For instance, the resource could be a processor or a point-to-point communication link. A given job, J_k, has the same relative priority across all resources in the distributed system. Different jobs require processing at a different sequence of nodes in the distributed system, and may have different start and end nodes.

Since jobs may revisit nodes, it is useful to differentiate between nodes and *stages* visited by a job. A stage is simply an instance of visiting a node. For example, a job that visits nodes 1, 2, then 1 is said to have a sequence of three stages, during which it visits the aforementioned nodes. Let the sequence of stages traversed by job J_k be called its *path*, p_k. In a departure from our previously published models [38, 42, 39], the union of paths traversed by all jobs *may contain loops*. For example, a job can revisit a node, or two jobs can visit two nodes in different orders. We therefore say that the path of job J_k contains one or more *folds*. A fold of J_k starting at node i is the largest sequence of nodes (in the order traversed by job J_k) that does not repeat a node twice. The first fold on path p_k starts with the first node that J_k visits. We denote the x^{th} fold of job J_k by J_k^x. For instance, if J_k has the path $(1, 2, 3, 1, 5, 6, 2)$, it is said to have two folds, namely $(1, 2, 3)$ and $(1, 5, 6, 2)$, denoted by J_k^1 and J_k^2 respectively. If the path of a job is acyclic, then it has only one fold that contains the whole path. The intuition for defining folds is that when jobs revisit a node multiple times, they may delay other jobs more than once on the same stage. In contrast, a single fold (of a job) can delay other jobs at most once per stage. Hence, folds will simplify the presentation of our proof. We denote the set of all folds of job J_k by Q_k.

Each job J_k must complete execution on all stages along its path p_k within its prespecified end-to-end deadline. The union of all the job paths forms a task graph. An arc in the task graph represents the direction of execution flow of a job, yielding a precedence constraint between the execution of the sub-jobs at the head and tail nodes of the arc. Observe that the task graph may contain cycles even if all jobs had one fold each. For example, consider a system of two jobs that traverse a sequence of nodes in opposite directions, such as the one shown in Figure 1. The task graph for this system contains a loop, as shown in Figure 5.1, even though individual jobs do not. Hence, loops in the task graph capture cyclic dependencies that may involve one or more jobs.

Let $C_{k,j}$ denote the worst-case execution time of job J_k on stage j in its path, and let D_k denote the relative end-to-end deadline of job J_k.

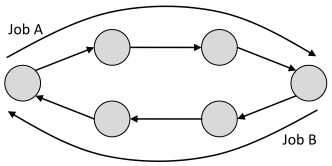

Figure 5.1: An task graph with a cycle

5.2 Delay in Non-Acyclic Task Graphs

In this section, we present the derivation of a worst-case end-to-end delay bound for a job in a distributed system with loops in the task graph under preemptive scheduling. This derivation enables construction of compact schedulability tests to determine if the system is schedulable. Towards the end of this section, we state the delay bound when the scheduling is non-preemptive, and omit the proof in the interest of brevity.

Let all jobs be numbered in priority order such that larger integers denote higher priority. When analyzing the delay of a job, since scheduling is preemptive and there is no blocking in our model, lower priority jobs can be ignored. Hence, without loss of generality, let the job whose end-to-end delay we wish to bound be denoted by J_1. This job executes along a path p_1 in the system, where p_1 may contain one or more folds.

We ignore the precedence constraints between successive folds of each higher priority job J_i, where $i > 1$. Thus, each fold of a higher priority job J_i becomes an independent job. We denote the xth fold of job J_i by (job) J_i^x. We call this process *unfolding*. Observe that unfolding does not eliminate cycles in the task graph because different folds of the same or different jobs can still visit nodes in different orders. Unfolding ensures, however, that job J_1 is delayed by any one higher priority fold at most once per J_1's stage.

It is easy to show that unfolding cannot decrease the delay of job J_1. Hence, if J_1 is schedulable after unfolding, then it is schedulable in the original job set. This is because unfolding merely removes some of the (precedence) constraints between stages of higher priority jobs. Hence, it increases the set of feasible higher-priority task arrival patterns that one needs to consider. A bound on J_1's delay computed by maximization over the larger set of possible arrival patterns can only be larger than one computed by maximization over the subset that respects the removed constraints (thus erring on the safe side). In the following, we therefore consider the unfolded job set when analyzing the delay of J_1.

Note that, a fold J_i^x can only preempt or delay J_1 when it shares a common execution node or a common sequence of nodes with J_1. Let us define a job segment $J_i^{x,s}$ as J_i^x's execution on a sequence of consecutive nodes on the path of J_i^x that is also traversed by J_1 either in the same order or exactly in reverse order. Let Seg_i^x be the set of all such segments for J_i^x. For example, if J_1 has the path (1,2,1,3,8,11,13) and J_i^x has the path (1,3,19,13,11,8), then $Seg_i^x = \{J_i^{x,1}, J_i^{x,2}\}$, where $J_i^{x,1}$ is the part of J_i^x that executes on nodes $(1, 3)$, and $J_i^{x,2}$ is the part of J_i^x that executes on nodes $(13, 11, 8)$.

Consider J_1 and the job segments (segments for short) that delay or preempt its execution. Each such

segment falls in one of three categories:

- *Forward flow segments:* Those are segments that share a consecutive set of stages with J_1 and traverse them in the same direction.
- *Reverse flow segments:* Those are segments that share a consecutive set of stages with J_1 and traverse them in the opposite direction.
- *Cross flow segments:* Those are segments composed of only one node. For example, such a segment may result from intersection of the path of J_1 with the path of another job in one node.

Figure 5.2 shows an example where the path of J_1 traverses five stages. Higher-priority job segments that share parts of that path are indicated by arrows that extend across the stages they execute on, pointing in the direction of the flow of the segment. Cross-flow segments are indicated by vertical arrows at the node they execute on.

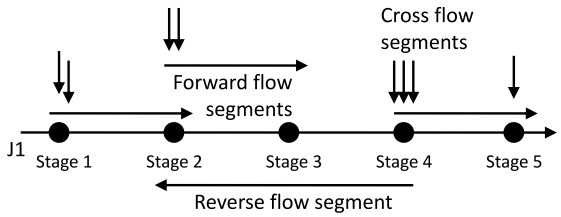

Figure 5.2: Three segment types

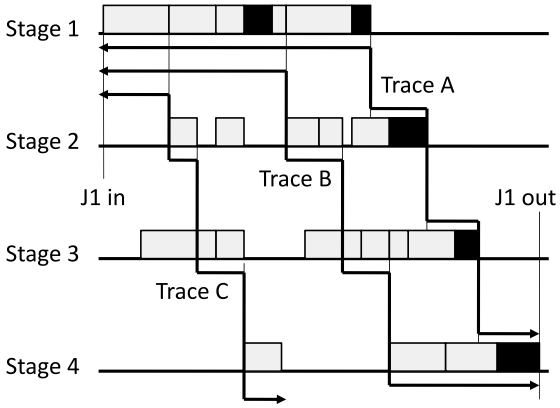

Figure 5.3: An execution trace

Consider the interval of time starting from the arrival time of J_1 to the system, until the finish time of J_1 on its last stage. The length of this interval is the end-to-end response time of J_1, which we wish to bound. Let us now define a *busy execution trace*, to mean a sequence of contiguous intervals of continuous processing on successive stages of path p_1 that collectively add up to the end-to-end delay of J_1. The intervals are contiguous in the sense that the end of a processing interval on one stage is the beginning of another processing interval on the next stage of path p_1. There may be many execution traces that satisfy the above definition. To reduce the number of different possibilities we further constrain the definition by requiring that each processing interval in the trace end at a job boundary (i.e., when some job's execution on that stage ends, which we shall call the job's *finish time* at that stage). Hence, the definition of a busy execution trace is as follows:

Definition 1: *A busy execution trace* through path p_1 is a sequence of contiguous intervals starting with

the arrival of J_1 on stage 1 and ending with the finish time of J_1 on the last stage of p_1, where (i) each interval represents a stretch of continuous processing on one stage, j, of path p_1, (ii) the interval on stage j ends at the finish time of some job on stage j, (iii) successive intervals are contiguous in that the end time of one interval on stage j is the start time of the next interval on stage $j+1$, and (iv) successive intervals execute on consecutive stages of path p_1.

Figure 5.3 presents examples of execution traces. In this figure, J_1, whose execution is indicated in black, traverses four stages, while being delayed and preempted by other jobs. The arrival time of J_1 to the first stage and its finish time on the last stage are indicated by J_1 *in* and J_1 *out*, respectively. Traces are depicted as staircase lines where the horizontal parts represent busy intervals at successive stages of p_1, and the vertical parts represent traversals to the next stage. *Trace A* and *Trace B*, in the figure, are examples of valid busy execution traces by our definition. *Trace C* does not satisfy the definition because it ends (i.e., runs into idle time) before the finish time of J_1 on the last stage. Remember that a busy execution trace, by definition, cannot contain idle time, since it is composed of contiguous intervals of continuous processing, ending with the finish time of J_1 on its last stage. In the following, we shall bound the length of a valid busy trace, hence, bounding the end-to-end response-time of J_1.

Observe that given work-conserving scheduling on all nodes, at least one busy execution trace always exists. Namely, it is the trace composed of the waiting intervals of J_1 on successive stages. This trace is indicated by *Trace A* in Figure 5.3. We call this trace the trace of *last traversal* because it ends its intervals on each stage at the finish time of the last (i.e., lowest priority) job. Let us now define the trace of *earliest traversal* as follows.

Definition 2: *A trace of earliest traversal* is a busy execution trace in which the end of an interval on stage j coincides with the finish time of the *first* job segment on stage j that (i) moves on to stage $j+1$ next, and (ii) shares at least one future stage $k > j$ with J_1, where both execute in the same busy period (or is J_1).

The second condition in the definition prevents construction of invalid traces, such as *Trace C* in Figure 5.3, that run into idle time before the completion of J_1 on the last stage. Because of that condition, one can show by induction that if starting at the first stage, there exists any valid execution trace from the current point on (which is always the case), then no stage traversal in the earliest traversal trace leads to a point that invalidates that property. Consequently, the trace of earliest traversal is always a valid trace.

Bounding the end-to-end delay of J_1 is equivalent to bounding the length of the trace of earliest traversal. First, we bound the amount of execution time that each types of job segments may contribute to the earliest traversal trace. For the purpose of expressing the aforementioned bound in a compact manner, it is convenient at this point to define $C_{i,max}^{x,s}$ to denote the maximum single-stage execution time of segment $J_i^{x,s}$ over its

joint path with J_1, and define $Node_{j,max}$ to denote the maximum stage execution time of all job-segments $J_i^{x,s}$ on node j. The three lemmas below bound delays due to the three types of segments depicted in Figure 5.2; namely, the forward flow segments, reverse flow segments and cross segments. We start with the most obvious ones first.

Lemma 1: A cross-flow segment, $J_i^{x,s}$, contributes at most one stage computation time to the length of the earliest traversal trace (bounded by $C_{i,max}^{x,s}$).

Proof. The lemma is trivially true since cross traffic segments, by definition, have only one stage. □

Lemma 2: A reverse-flow segment, $J_i^{x,s}$, contributes at most one stage computation time to the length of the earliest traversal trace (bounded by $C_{i,max}^{x,s}$).

Proof. The lemma is true because reverse flow segments execute on the nodes of the system in the reverse order from J_1. Since the earliest traversal trace follows the path of J_1, if $J_i^{x,s}$ was included in the interval of the trace at stage j, then it must have departed stage $j+1$ before the beginning of the interval of the trace on stage $j+1$. Similarly, it will arrive at stage $j-1$ after the end of the interval of the trace on stage $j-1$. □

Lemma 3: The total contribution of all forward-flow segments, $J_i^{x,s}$, to the length of the earliest traversal trace is bounded by:

$$\sum_{segments} C_{i,max}^{x,s} + \sum_{\substack{forward-flow \\ segments}} C_{i,max}^{x,s} + \sum_{j \in p_1} Node_{j,max} \tag{5.1}$$

Proof. Let us define the *end stage* of a forward-flow job segment as either its last stage or the stage after which it is always separated by idle time from J_1 (and hence need not be considered further), whichever comes first. It is convenient to partition the contribution of forward-flow segments to the length of the trace into (i) the total length due to stage execution times of segments at their end stages, denoted by C_{ff_1}, (ii) the total length due to stage execution times of segments that preempt other segments and execute ahead of lower priority segments that arrived earlier at the stage, denoted by C_{ff_2}, and (iii) the total length of stage execution times of segments not at their end stages, and that do not preempt another segment, denoted by C_{ff_3}.

To bound C_{ff_1}, the total length due to stage execution times of segments at their end stages, note that each forward-flow segment, $J_i^{x,s}$, has only one end stage. Its length is at most $C_{i,max}^{x,s}$. The total of all end-stage computation times over all segments is thus given by:

$$C_{ff_1} \leq \sum_{\substack{forward-flow \\ segments}} C_{i,max}^{x,s} \tag{5.2}$$

To bound, C_{ff_2}, note that, each segment can preempt another segment at most once along the earliest traversal trace. Consider a segment $J_i^{x,s}$ that preempts another segment in the earliest traversal trace at stage j. By definition of the earliest traversal trace (see Definition 2), starting from the time this preemption occurs, no segment of lower priority than $J_i^{x,s}$ can be a part of the earliest traversal trace until the end stage of $J_i^{x,s}$. Therefore, $J_i^{x,s}$ will not preempt any other segment in the earliest traversal trace. Thus, the total length of stage execution times of segments in the earliest traversal trace that preempt and execute ahead of lower priority segments that arrived earlier is bounded by:

$$C_{ff_2} \leq \sum_{segments} C_{i,max}^{x,s} \tag{5.3}$$

To bound C_{ff_3}, observe that, there exists at most one execution time of a segment at each stage of the earliest traversal trace that is not an end stage of a segment and that does not execute ahead of a lower priority segment that arrived earlier in the earliest traversal trace (that is, not bounded by C_{ff_1} or C_{ff_2}). Let us assume the contrary and suppose that there are two execution times of segments J_i and J_k at a stage j in the earliest traversal trace that are not included in C_{ff_1} or C_{ff_2}. Without loss of generality, let us also assume that J_i arrives at stage j before J_k. Now, J_k cannot be a higher priority segment that arrives after J_i and completes execution before J_i (covered under C_{ff_2}). Thus, J_k can start executing on stage j only after J_i completes execution. As stage j is not the end stage of J_i, by definition, the portion of the earliest traversal trace on stage j should end with the execution of J_i and cannot include the execution of J_k, resulting in a contradiction. Therefore, there exists at most one execution time of a segment at each stage of the earliest traversal trace that is not bounded under C_{ff_1} or C_{ff_2}. Thus,

$$C_{ff_3} \leq \sum_{j \in p_1} Node_{j,max} \tag{5.4}$$

Adding up C_{ff_1}, C_{ff_2} and C_{ff_3}, given by Equations (5.2), (5.3), and (5.4), the lemma follows. □

Consider all jobs J_i, each made of a set of folds, denoted by Q_i, where each fold $J_i^x \in Q_i$ gives rise to one or more segments, $J_i^{x,s}$, collectively called set Seg_i^x. The following theorem presents the delay bound on J_1 in the system.

Preemptive Delay Composition Theorem. *For a preemptive, work-conserving scheduling policy that assigns the same priority across all stages for each job, and a different priority for different jobs, the end-to-end delay of a job J_1 following path p_1 can be composed from the execution parameters of higher priority jobs that delay or preempt it as follows:*

$$Delay(J_1) \leq \sum_i \sum_{J_i^x \in Q_i} \sum_{J_i^{x,s} \in Seg_i^x} 2C_{i,max}^{x,s} + \sum_{j \in p_1} Node_{j,max} \tag{5.5}$$

Proof. The theorem follows trivially from Lemma 1, 2, and 3, by adding the contributions of all cross-flow, reverse-flow, and forward-flow segments to the trace. □

We shall now state the theorem under non-preemptive scheduling, but omit its proof. Let $Node_{j,all\_max}$ denote the maximum computation time of any job (not just higher priority jobs) on stage j, and let $Node_{j,lower\_max}$ denote the maximum computation time of any lower priority job that joins the path of J_1 on stage j.

Non-preemptive Delay Composition Theorem. *For a non-preemptive scheduling policy that assigns the same priority across all stages for each job, and a different priority for different jobs, the end-to-end delay of a job J_1 following path p_1 can be composed from the execution parameters of jobs that delay it as follows:*

$$Delay(J_1) \leq \sum_i \sum_{J_i^s \in Q_i} \sum_{J_i^{x,s} \in Seg_i^x} C_{i,max}^{x,s} + \sum_{j \in p_1} Node_{j,all\_max} + \sum_{j \in p_1} Node_{j,lower\_max} \tag{5.6}$$

The above delay bound for any job can be calculated in $O(MN)$ time, where N is the number of stages in the system and M is the number of tasks. Each higher priority task's path can be broken down into various segments and the maximum computation time for the task on each of its segments can be calculated in $O(N)$ time. This has to be repeated for at most M tasks. Likewise, the maximum computation time of higher priority tasks on a stage, $Node_{j,max}$, can be calculated in $O(M)$ time and this needs to be repeated for at most N stages. Therefore, the net complexity of calculating the delay bound is $O(MN)$. In contrast, existing techniques to calculate the end-to-end delay bound for tasks such as holistic analysis and network calculus, have a pseudo-polynomial time complexity as they involve an iterative solution until convergence is reached.

5.3 Schedulability Analysis

Using the end-to-end delay bound for non-acyclic systems derived in the previous section, we can reduce the schedulability analysis of tasks in a distributed system with cycles to that of analyzing an equivalent hypothetical uniprocessor, similar to the technique presented in Section 3.4. To analyze the schedulability of a job J_1, the transformation is carried forth as follows:

- Each higher priority job-segment $J_i^{x,s}$ in the distributed system, is replaced by a uniprocessor job $J_i^{x,s*}$ with computation time equal to $2C_{i,max}^{x,s}$ and same deadline as J_i;
- Job J_1 is replaced by a uniprocessor job J_1^* with computation time equal to $C_{1,max} + \sum_{j \in p_1} Node_{j,max}$ and deadline same as J_1

Hence, if the uniprocessor job J_1^* is schedulable, so is job J_1 in the original distributed system. In the case of periodic tasks, uniprocessor jobs which are invocations of the same periodic task can be grouped together to form a periodic task on the uniprocessor. When the end-to-end deadlines of tasks are larger than the period, then for each higher priority task T_i we need to account for the task invocations that can be present in the system when J_1 arrives, which can be bounded by $\lceil D_i/P_i \rceil$. Further, if the task T_1 being analyzed has cycles in its path, then earlier invocations of T_1 may delay invocations that arrive later. Therefore, T_1 also needs to be included in the set of higher priority tasks. When the end-to-end deadline of tasks is lesser than the period, then T_1 need not be included as a higher priority task when analyzing its schedulability.

The end-to-end delay bound for non-acyclic systems derived in this chapter, thus enables any uniprocessor schedulability test to be used to analyze the schedulability of jobs in the distributed system. If tests such as the Liu and Layland test [60] for periodic tasks is used as the uniprocessor test, then closed-form expressions can be derived for analyzing the schedulability of tasks in distributed systems that contain cycles.

5.4 An Illustrative Example

In this section, we shall illustrate using a simple example, as to how the bound derived in this chapter can result in tighter end-to-end delay estimates for non-acyclic task systems. We consider a system consisting of four nodes or stages, namely S_1, S_2, S_3, and S_4. We consider two tasks, T_1 and T_2, with T_2 having a higher priority than T_1. Let the period equal to the end-to-end deadline of T_2 be 10 units, and that of T_1 be 12 units. Task T_2 follows the path $S_1 - S_2 - S_3 - S_4$, and T_1 follows the path $S_1 - S_2 - S_3 - S_4 - S_3 - S_2 - S_1$, as shown in Figure 5.4. Let the sub-job of T_2 executing on stage j be denoted as $T_{2,j}$. The sub-jobs of T_1 are denoted as $T_{1,1}, T_{1,2}, \ldots, T_{1,7}$ in the order in which they execute. For simplicity, let us assume that the computation times for each task on every stage is one unit. The objective is to estimate the end-to-end delay and schedulability of T_1.

Let us first analyze the system using holistic analysis [89]. The response time for each sub-task is at least as large as the computation time. So, the initial response times $R_{1,j}^0 = 1$, and the jitter for all sub-jobs is set to zero $J_{1,j}^0 = 0$. We now start the iterative process of estimating new response times, and updating the response times based on the jitter values. In the first iteration, each sub-job of T_1 is delayed by one invocation of T_2. Also, $T_{1,1}$ and $T_{1,7}$ interfere with each other as they execute on the

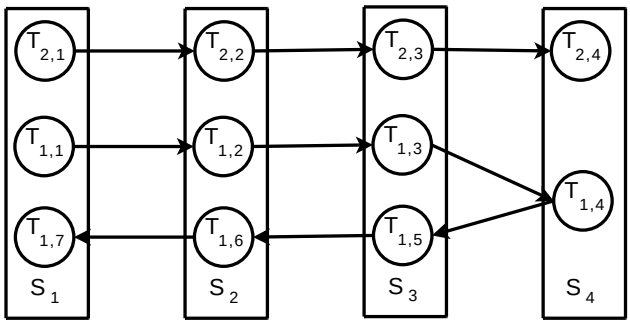

Figure 5.4: Figure showing the paths followed by the tasks T_1 and T_2 in the example

same node (likewise, $T_{1,2}$ and $T_{1,6}$, $T_{1,3}$ and $T_{1,5}$ interfere with each other). Let us assume that a sub-job with a lower index has a higher priority. We therefore obtain $R^1_{1,1} = R^1_{1,2} = R^1_{1,3} = R^1_{1,4} = 2$, and $R^1_{1,5} = R^1_{1,6} = R^1_{1,7} = 3$ (these sub-jobs are delayed by T_2 and the lower index sub-job of T_1). We now update the jitter values as the sum of the jitter and response-time of the sub-job executing on the previous stage. That is, $J^1_{1,1} = 0, J^1_{1,2} = 2, J^1_{1,3} = 4, J^1_{1,4} = 6, J^1_{1,5} = 8, J^1_{1,6} = 11, J^1_{1,7} = 14$. We need to follow this iterative process until convergence, but even at the first iteration the end-to-end response time of T_1 exceeds its end-to-end deadline, and T_1 is declared unschedulable. One can see that this process will quickly lead to the end-to-end response time to blow up for large systems.

Improvements to holistic analysis have been presented in [68, 73], that use the notion of *offset* instead of jitter. One problem with holistic analysis is that by assuming the response time at a stage to be the jitter for the next stage, the jitter values increase with longer path lengths. To overcome this problem, [68, 73] set the response time at a stage to be the offset for the next stage. The offset value denotes the minimum time after which the sub-job is activated. This makes the analysis more accurate, but more complicated as well. Using this analysis, we can obtain the response times for the sub-jobs of T_1 in the first iteration as $R_{1,j} = 2$, for $j = 1..7$. Here again we need to perform an iterative process until convergence, but just the first iteration tells us that the end-to-end response time estimate of 14 units for T_1 from this analysis also exceeds the end-to-end deadline of 12 units.

The fundamental problem with the above analysis is that T_2 delays a sub-job of T_1 at every stage along its path from stage S_1 to S_4 (the response time of each sub-job is calculated as 2 units). However, in reality this is not the case. When an invocation of T_2 delays an invocation of T_1 at stage S_1, as it has the highest priority, it will execute on future stages without waiting and hence will never delay T_1 on the remaining stages. By analyzing the system one stage at a time, existing analysis techniques fail to accurately account for the parallelism in the execution of different stages in the distributed system. Now, let us analyze the schedulability of T_1 based on the end-to-end delay bound derived in this chapter. As the end-to-end deadline of T_1 is not larger than the period, we do not have to include T_1 in the set of higher priority tasks. So, T_2

is the only higher priority task and has only one segment with T_1. We therefore create a uniprocessor task T_2^* with a computation time of 2 units (twice the maximum stage execution time) and period of 12 units. We construct a task T_1^* with a computation time of $1 + 7 = 8$ time units (its own computation time of 1 unit and the sum of the maximum execution times of any job at each of the seven stages along the path of T_1). Using the response time analysis test for the hypothetical uniprocessor [8], we obtain the worst-case end-to-end response time of T_1 as $8 + 2 = 10$ units. Thus, T_1 is found to be schedulable in the original distributed system. By analyzing the system as a whole, the end-to-end delay bound derived in this chapter is able to provide a more accurate bound on the end-to-end delay of tasks in distributed systems with cycles in the task graph.

5.5 Evaluation

In this section, we evaluate the end-to-end delay bound for non-acyclic systems using simulation studies for periodic tasks. We compare it with three other analysis techniques. We call the first the traditional test, that breaks the end-to-end deadline of each task into per-stage deadlines and analyzes each stage independently. If all per-stage deadlines are met then the system is deemed to be schedulable. The second test is holistic analysis applied to non-acyclic systems [89], that uses an iterative procedure to converge to worst-case response time values at each stage for every task. The third test is based on our own previous work for acyclic systems, by cutting any cycles in the system and relaxing precedence constraints (as discussed in Section 4.8). We do not compare with network calculus [18, 19] or its extensions such as [49], as the solution to handle cycles in the task graph requires that a system of simultaneous equations be constructed, and it may be difficult or even impossible to obtain delay bounds for certain scenarios. Further, previous comparisons such as [52] have found holistic analysis to perform better than network calculus approaches. For each test we construct an admission controller that would admit as many tasks as it can deem feasible, and measure the average per stage (resource) utilization achieved.

The schedulability test used is assumed to be deadline monotonic scheduling. We consider two types of non-acyclic traffic. The first reflects request-response type traffic, where the request follows a sequence of execution nodes, and the response follows the same set of nodes but in the opposite direction. The second traffic type emulates web server requests, where each task follows a sequence of nodes from S_1 to S_n and returns in the opposite direction from S_{n-1} to S_1. Thus, each task executes twice at each stage except S_n, once in the forward direction and once in the reverse direction. Note that in the second traffic type, each task's path contains cycles, whereas in the first scenario, the task paths are acyclic, but with tasks going in opposite directions. The larger jitter values due to the presence of cycles in each task's paths causes holistic

analysis to perform worse in the second scenario (as observed in our simulation studies below), although the two traffic types are seemingly similar to one another.

Simulation parameters are chosen similar to those in Section 3.7. The default value of the deadline ratio parameter, DR, is assumed to be 2.0. The default value of the task resolution parameter, T, is chosen as 1/50. The response time analysis technique presented in [8] is used as the schedulability test for the composed hypothetical uniprocessor.

Each point in the figures below represent average values obtained from 100 executions, with each execution consisting of 80000 task invocations. For the purpose of comparing different admission controllers, each admission controller was allowed to execute on the same 100 task sets. The 95% confidence interval for all the values presented is within 1% of the mean value, and is not plotted for the sake of legibility.

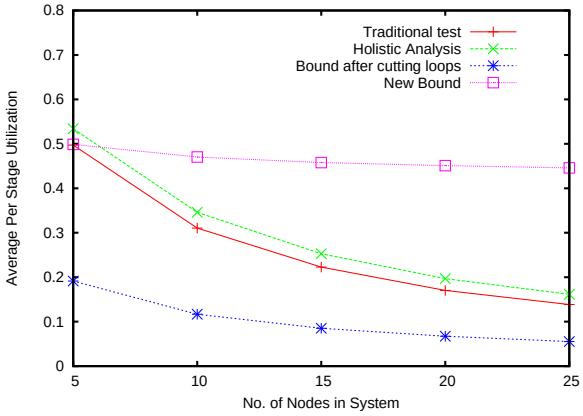

Figure 5.5: Comparison of average per stage utilization for different number of stages in the system for request-response type traffic

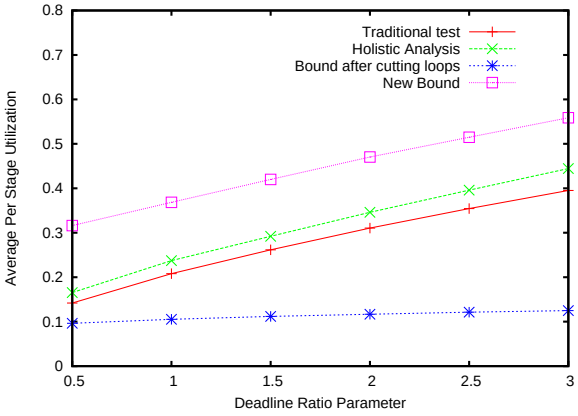

Figure 5.6: Comparison of average per stage utilization for different deadline ratio parameter values for request-response type traffic

In Figure 5.5, we compare the average per-stage utilization of the four schedulability tests for different number of nodes in the system for request-response type traffic. So, for each task there are other tasks that traverse the system in the same direction as well as in the opposite direction. The end-to-end delay bound presented in this chapter is able to ensure nearly the same per-stage utilization regardless of the number of stages in the system. In contrast, all the other tests become increasingly pessimistic with system scale. The acyclic bound after cutting loops performs poorly as for each job that traverses the system in the opposite direction, the cycles are broken by cutting the job at every link creating N independent sub-jobs. These sub-jobs can therefore arrive independently of each other in a worst-case manner so as to delay the lower priority job at every stage. Holistic analysis and the traditional test analyze the system one stage at a time and fail to accurately account for the parallelism in the execution of different stages. For large systems, the jitter for downstream sub-jobs becomes large as the jitter increases with increasing number of nodes in the task path, causing holistic analysis to perform poorly for large system sizes.

For the same traffic pattern, for a system of 10 stages, we vary the deadline ratio parameter and plot the results in Figure 5.6. A larger value of the deadline ratio parameter implies that the range of deadline values is larger. This allows lower priority tasks with large deadlines to execute in the background of higher priority tasks with shorter deadlines, increasing the overall utilization of the system. This trend is observed for all the four schedulability tests. The new bound significantly outperforms the other tests for all deadline ratio parameter values.

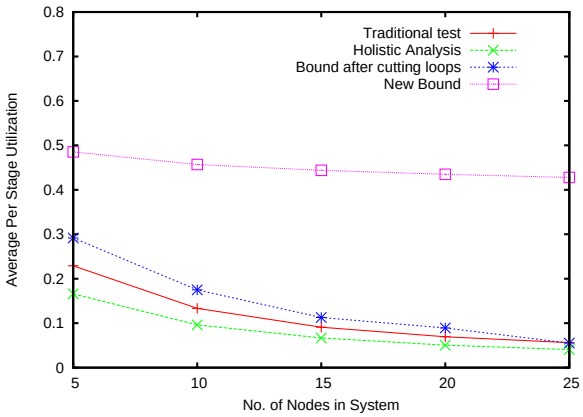

Figure 5.7: Comparison of average per stage utilization for different number of stages in the system for web server type traffic

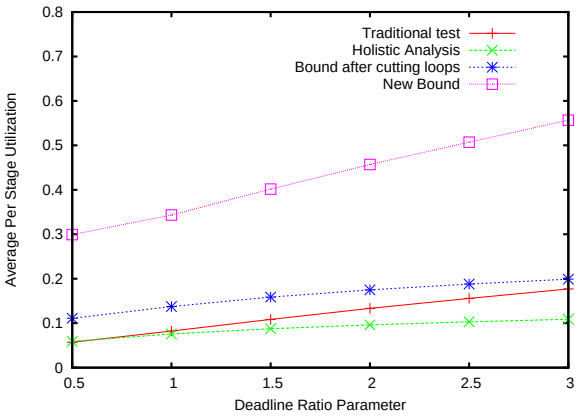

Figure 5.8: Comparison of average per stage utilization for different deadline ratio parameter values for web server type traffic

For web server type traffic, where each task traverses the stages in the system in the forward direction and then in the reverse direction, we plot the average per stage utilization for increasing number of stages in the system in Figure 5.7. As observed in Figure 5.5, the new bound is able to achieve nearly the same per stage utilization regardless of system size. Also note that holistic analysis and the traditional test perform poorly for this traffic scenario compared to their achieved utilization under request-response type traffic shown in Figure 5.5. For holistic analysis, the jitter values increase considerably due to the presence of cycles in the task path and the large path length causing the analysis to be extremely pessimistic. The traditional test breaks the end-to-end deadline into per-stage deadlines, which works poorly when the path length is long, as the delay experienced by tasks at different stages is not uniform.

Figure 5.8, presents a comparison of the four schedulability tests for different deadline ratio parameter values for the web server type traffic scenario in a system with 10 stages. As observed in Figure 5.7, holistic analysis and the traditional test perform poorly for this traffic scenario. The utilization values are observed to increase with increasing deadline ratio parameter values, as low priority jobs with large deadlines are able to execute in the background of higher priority jobs with short deadlines, thereby increasing the overall utilization of each stage. The new bound significantly outperforms the other schedulability tests for such systems with long path lengths.

Chapter 6

Delay Composition Algebra

In this chapter, we present an algebra for schedulability analysis of distributed real-time systems. The algebra reduces the distributed system workload to an equivalent uniprocessor workload that can be analyzed using uniprocessor schedulability analysis techniques to infer end-to-end delay and schedulability properties of each of the original distributed jobs. The reduction is carried out for all the jobs in the system simultaneously, without having to repeat the reduction using the delay composition theorem to analyze the schedulability of each job. The rest of this chapter is organized as follows. We present the algebra and the intuition behind it in Section 6.1. In Section 6.2, we formally prove the correctness of the algebra. In Section 6.3, we evaluate the performance of our algebraic framework through simulation studies.

6.1 Delay Composition Algebra

The main goal of the delay composition algebra is to allow schedulability analysis of distributed jobs by reducing them to a uniprocessor workload. Given a graph of system resources, where nodes represent processing resources and arcs represent the direction of job flow, our algebraic operators systematically "merge" resource nodes, composing their workloads per rules of the algebra, until only one node remains. The workload of that node represents a uniprocessor job set. Uniprocessor schedulability analysis can then be used to determine the schedulability of the set. In this section, we provide a detailed description of the algebra and its underlying basic intuition.

We consider arbitrary non-acyclic systems (the system model being similar to Section 5.1). We further augment the DAG with an arc from each end node of a job to a single virtual "finish" node, f. The execution time of any job J_i on the finish node f, $C_{i,f}$, is set to zero, so as to not affect schedulability. This augmentation ensures that the graph is never partitioned and hence can be reduced to a single node using our algebraic operators. The question we would like to answer is whether each job is schedulable (i.e., can traverse its path through the system by its deadline).

We provide the intuition leading to our algebra in Section 6.1.1. In Section 6.1.2, we describe the basic operand representation and show how to translate a system into operands of the delay composition algebra.

The operators of the algebra and a proof of liveness are described in Section 6.1.3. In Section 6.1.4, we show how end-to-end delay and schedulability of jobs are determined from the final operand matrix. Finally, we conclude with an illustrative example, in Section 6.1.5.

6.1.1 Intuition for a Reduction Approach

To answer the schedulability question, we reduce the distributed system to a single node. Our reduction operators simplify the resource graph progressively by breaking forks into chains and compacting chains by merging neighboring nodes, producing an equivalent workload for the resulting merged node. Workload of any one node (that may represent a single resource or the result of reducing an entire subsystem) is described generically by a two-dimensional matrix stating the worst-case delay that each job, J_i, imposes on each other job, J_k, in the subsystem the node represents. Let us call it the *load matrix* of the subsystem in question.

Observe that if jobs are invocation instances of periodic or sporadic tasks (which we expect to be the most common use of our algebra), we include in the load matrix only one instance of each task. We need to consider only one instance of each task because all individual invocation instances of the same task have the same parameters and thus will impose the same delay on a lower priority instance. It is therefore enough to compute this delay once. We are able to get away with this because our algebra is only concerned with job transformation. It is not concerned, for example, with computing the number of invocations of one task that may preempt another. This is the responsibility of uniprocessor schedulability analysis that we apply to the resulting uniprocessor task set. The algebra simply reduces a distributed instance into a uniprocessor instance. This decoupling between the reduction part and the analysis part is a key advantage of the reduction-based approach. Hence, in the following, when we mention a job, it could either mean an aperiodic job or a single representative instance of a periodic or sporadic task. For periodic or sporadic task sets, the dimension of the load matrix is therefore $n \times n$, where n is the finite number of *tasks* in the set.

Observe that, on a node that represents a single resource j, any job J_i, that is of higher priority than job J_k, can delay the latter by at most J_i's worst-case computation time, $C_{i,j}$, on that resource. This allows one to trivially produce the load matrix for a single resource given job computation times, $C_{i,j}$, on that resource. Element (i, k) of the load matrix for resource j, denoted $q^j_{i,k}$ (or just $q_{i,k}$ where no ambiguity arises) is equal to $C_{i,j}$ as long as J_i is of (equal or) higher priority than J_k. It is zero otherwise.

The main question becomes, in a distributed system, how to compute the worst-case delay that a job imposes on another when the two meet on more than one resource? The answer decides how delay components of two load matrices are combined when the resource nodes corresponding to these matrices are merged using our algebraic operators. Intuitions derived from uniprocessor systems suggest that delays are combined

additively. This is not true in distributed systems. In particular, as shown in Chapter 3, delays in pipelines are *sub-additive* because of gains due to parallelism caused by pipelining. More specifically, the worst-case delay imposed by a higher priority job, J_i, on a lower priority job, J_k, when both traverse the same set of stages, varies with the *maximum* of J_i's per-stage computation times, not their *sum* (plus another component we shall mention shortly).

The delay composition algebra leverages the aforementioned result. Neighboring nodes in the resource graph present an instance of pipelining, in that jobs that complete execution at one node move on to execute at the next. Hence, when these neighboring nodes are combined, the delay components, $q_{i,k}$, in their load matrices are composed by a maximization operation. In our algebra, this is done by the PIPE operator. It reduces two neighboring nodes to one and combines the corresponding elements, $q_{i,k}$, of their respective load matrices by taking the maximum of each pair. For this reason, we call $q_{i,k}$ the *max term*.

It could be, however, that two jobs travel together in a pipelined fashion[1] for a few stages (which we call a pipeline segment), then split and later merge again for several more stages (i.e., another pipeline segment). Figure 6.1 demonstrates such a scenario for a job J_k and a higher priority job, J_i. In this case, the max terms of each of the pipeline segments (computed by the maximization operator) must be added up to compute the total delay that J_i imposes on J_k. It is convenient to use a running counter or "accumulator" for such addition. Whenever the jobs are pipelined together, delays are composed by maximization (kept in the max term) as discussed above. Every time J_i splits away from J_k, signaling the termination of one pipeline segment, the max term (i.e., the delay imposed by J_i on J_k in that segment) is added to the accumulator. Let the accumulator be denoted by $r_{i,k}$. Hence, $r_{i,k}$ represents the total delay imposed by J_i on a lower priority job J_k over all past pipeline segments they shared. Observe that jobs can split apart only at those nodes in the DAG that have more than one outgoing arc. Hence, in our algebra, a SPLIT operator is used when a node in the DAG has more than one outgoing arc. SPLIT updates the respective accumulator variables, $r_{i,k}$, of all those jobs J_k, where J_k and a higher priority job J_i part on different arcs. The update simply adds $q_{i,k}$ to $r_{i,k}$ and resets $q_{i,k}$ to zero.

In summary, in a distributed system, it is useful to represent the delay that one job J_i imposes on another J_k as the sum of two components $q_{i,k}$ and $r_{i,k}$. The $q_{i,k}$ term is updated upon PIPEs using the maximization operator (the max term). The $r_{i,k}$ is the accumulator term. The $q_{i,k}$ is added to the $r_{i,k}$ (and reset) upon SPLITs, when J_i splits from the path of J_k. PIPE and SPLIT are thus the main operators of our algebra. In the final resulting matrix, the $q_{i,k}$ and $r_{i,k}$ components are added to yield the total delay that each job

[1]The term *pipelined execution* has also been used in the literature to refer to the situation where an invocation of a task can start before the previous invocation has completed, when deadlines are larger than task periods. We do not intend the term pipelined execution in this context.

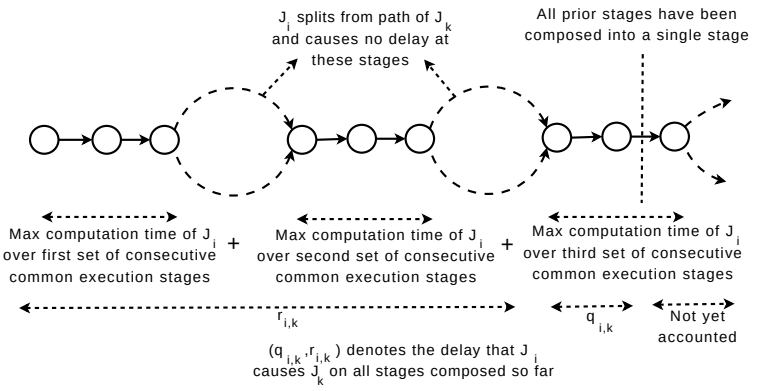

Figure 6.1: Figure showing the components of the delay that J_i causes J_k, and how the composition of stages works

imposes on another in the entire system.

The final matrix is indistinguishable from one that represents a uniprocessor task set. In particular, each *column* k in the final matrix denotes a uniprocessor set of jobs that delay J_k. In this column, each non-zero element determines the computation time of one such job J_i. Since the transformation is agnostic to periodicity, for periodic tasks, J_i and J_k simply represent the parameters of the corresponding periodic task invocations. Hence, for any task, T_k, in the original distributed system, the final matrix yields a uniprocessor task set (in column k), from which the schedulability of task T_k can be analyzed using uniprocessor schedulability analysis.

Finally, the above discussion omitted the fact that the results in Chapter 3 also specified a component of pipeline delay that grows with the number of stages traversed by a job and is independent of the number of higher priority jobs. We call it the *stage-additive* component, s_k. Hence, the load matrix, in fact, has an extra row to represent this component. As the name suggests, when two nodes are merged, this component is combined by addition. With the above background and intuition in mind, in the following subsections, we describe the algebra more formally, then prove it.

6.1.2 Operand Representation

In order to represent a task set on a resource for the purpose of analyzing delay and schedulability, we represent the delay that each job (or periodic task invocation) causes every other job in the system. As mentioned above, we represent this as an $n \times n$ array of delay terms, with the $(i,k)^{th}$ element denoting the delay that job J_i causes job J_k. Each element (i,k) is represented as a two-tuple $(q_{i,k}, r_{i,k})$, where the first term in the tuple $q_{i,k}$ denotes the max-term, and the second term $r_{i,k}$ denotes the accumulator-term. The operand matrix has an additional row in which the k^{th} element, s_k (that we shall define shortly), represents the delay of job J_k that is independent of the number of jobs in the system, and is additive across the stages

on which J_k executes. An operand A, represented as an $(n+1) \times n$ matrix is shown below:

$$A = \left(\begin{array}{c|cccc} & J_1 & J_2 & . \ . & J_n \\ \hline J_1 & (q^A_{1,1}, r^A_{1,1}) & (q^A_{1,2}, r^A_{1,2}) & . \ . & (q^A_{1,n}, r^A_{1,n}) \\ J_2 & (q^A_{2,1}, r^A_{2,1}) & (q^A_{2,2}, r^A_{2,2}) & . \ . & (q^A_{2,n}, r^A_{2,n}) \\ . & . & & & \\ . & . & & & \\ J_n & (q^A_{n,1}, r^A_{n,1}) & (q^A_{n,2}, r^A_{n,2}) & . \ . & (q^A_{n,n}, r^A_{n,n}) \\ & \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots & & & \\ & s^A_1 & s^A_2 & . \ . & s^A_n \end{array}\right)$$

Let us now construct the matrix for a single stage j, the basic operand. Let us first assume that the scheduling is preemptive. Without loss of generality, let jobs be indexed in order of priority and $i < k$ imply that J_i has a higher priority than J_k. Consider a job J_k and the column corresponding to it. The accumulator term $r_{i,k}$ is set to zero, for all i. If J_k does not execute at stage j, then $q_{i,k}$ and s_k are set to zero, for all i. If J_k executes at stage j, but a job J_i does not or if it has a lower priority than J_k, then $q_{i,k}$ is set to zero. If J_i executes on stage j exactly once, then $q_{i,k}$ is set to $C_{i,j}$. If J_i visits stage j multiple times, then $q_{i,k}$ is set to the maximum computation time of J_i over all its visits to stage j. The stage-additive component, s_k is defined as the maximum computation time of any higher priority job on stage j, counted as many times as J_k visits the stage. Suppose that J_k visits the stage p times, then $s_k = p \times \max_{i \leq k} C_{i,j}$. An example operand matrix for a stage j in a system with four jobs, of which only J_1, J_2 and J_4 execute on the stage, is shown below:

$$\left(\begin{array}{c|cccc} & J_1 & J_2 & J_3 & J_4 \\ \hline J_1 & (C_{1,j}, 0) & (C_{1,j}, 0) & (0,0) & (C_{1,j}, 0) \\ J_2 & (0,0) & (C_{2,j}, 0) & (0,0) & (C_{2,j}, 0) \\ J_3 & (0,0) & (0,0) & (0,0) & (0,0) \\ J_4 & (0,0) & (0,0) & (0,0) & (C_{4,j}, 0) \\ & \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots & & & \\ & C_{1,j} & max(C_{1,j}, C_{2,j}) & 0 & max(C_{1,j}, C_{2,j}, C_{4,j}) \end{array}\right)$$

Under non-preemptive scheduling, the matrix is constructed in a very similar manner, except for the stage-additive component s_k, which is defined as the sum of two terms. The first term is the maximum computation time of any job (not just higher priority jobs) on stage j, and the second term is the maximum computation

time of any lower priority job on stage j, each counted p times. That is, $s_k = p(\max_i C_{i,j} + \max_{i>k} C_{i,j})$. An example matrix for a stage j under non-preemptive scheduling, for the same 4-job system as before is shown below:

$$\left(\begin{array}{c|cccc} & J_1 & J_2 & J_3 & J_4 \\ \hline J_1 & (C_{1,j},0) & (C_{1,j},0) & (0,0) & (C_{1,j},0) \\ J_2 & (0,0) & (C_{2,j},0) & (0,0) & (C_{2,j},0) \\ J_3 & (0,0) & (0,0) & (0,0) & (0,0) \\ J_4 & (0,0) & (0,0) & (0,0) & (C_{4,j},0) \\ & \multicolumn{4}{c}{\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots} \\ & C_{1,j}+ & max(C_{1,j}, & 0 & max(C_{1,j}, \\ & max(C_{2,j},C_{4,j}) & C_{2,j})+C_{4,j} & & C_{2,j},C_{4,j}) \end{array}\right)$$

6.1.3 Operators of the Algebra

We describe the two operators, namely PIPE and SPLIT. These operators ensure that every term $(q_{i,k}, r_{i,k})$ in the resultant operand matrix correctly represents the max-term and accumulator-term of the delay that J_i can cause J_k over all the stages that the operand represents.

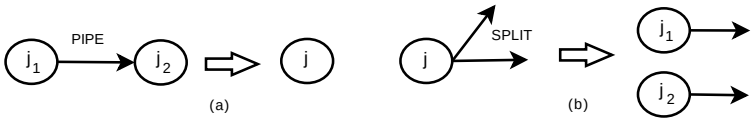

Figure 6.2: Figure showing the operators and the equivalent stages they result in (a) PIPE (b) SPLIT

The PIPE Operator

The PIPE operator merges two neighboring nodes in the resource graph (as shown in Figure 6.2(a)). Each of the two nodes being merged may themselves be resulting from the composition of multiple nodes. PIPE can be applied to any two nodes connected by an arc as long as the node at the tail of the arc (i.e., the upstream node) has only one outgoing arc. If the node has more than one outgoing arc, it must be split first as described in the SPLIT operator.

Let $C = A\ PIPE\ B$, where A, B, and C are matrices of the form described in Section 6.1.2. The result of the PIPE operation $(q^C_{i,k}, r^C_{i,k})$ is obtained by taking the maximum of corresponding elements $q_{i,k}$ and $r_{i,k}$ from the two operand matrices A and B. As we shall show later in Section 6.2, only the first (i.e., upstream) of the elements $r_{i,k}$ from the two operand matrices can be non-zero, so the max operation on the $r_{i,k}$ elements essentially copies the upstream value of $r_{i,k}$ onto matrix C. The stage-additive component, s^C_k, on the other hand is additive across stages, and hence the corresponding stage-additive components from the two operand matrices are added. The PIPE operator can formally be defined as follows:

Definition 1: PIPE Operator. For *any* two neighboring nodes in the resource graph, represented by operand matrices A and B, if the upstream node has exactly one outgoing arc, the two nodes can be composed into a single node represented by matrix C using the PIPE operator, $C = A\ PIPE\ B$, as follows:

1. $\forall i,k$: $q^C_{i,k} = max(q^A_{i,k}, q^B_{i,k})$
2. $\forall i,k$: $r^C_{i,k} = max(r^A_{i,k}, r^B_{i,k})$
3. $\forall k$: $s^C_k = s^A_k + s^B_k$

□

For instance, when jobs J_1 and J_2 execute on stages 1 and 2 (represented as matrices A and B, respectively), the PIPE operation between the two stages can be denoted as:

$$\left(\begin{array}{c|cc} & J_1 & J_2 \\ \hline J_1 & (q^A_{1,1}, r^A_{1,1}) & (q^A_{1,2}, r^A_{1,2}) \\ J_2 & (0,0) & (q^A_{2,2}, r^A_{2,2}) \\ & \cdots\cdots\cdots\cdots & \\ & s^A_1 & s^A_2 \end{array}\right) PIPE \left(\begin{array}{c|cc} & J_1 & J_2 \\ \hline J_1 & (q^B_{1,1}, r^B_{1,1}) & (q^B_{1,2}, r^B_{1,2}) \\ J_2 & (0,0) & (q^B_{2,2}, r^B_{2,2}) \\ & \cdots\cdots\cdots\cdots & \\ & s^B_1 & s^B_2 \end{array}\right) = \left(\begin{array}{c|cc} & J_1 & J_2 \\ \hline J_1 & \begin{array}{l}(max(q^A_{1,1}, q^B_{1,1}),\\ max(r^A_{1,1}, r^B_{1,1}))\end{array} & \begin{array}{l}(max(q^A_{1,2}, q^B_{1,2}),\\ max(r^A_{1,2}, r^B_{1,2}))\end{array} \\ J_2 & (0,0) & \begin{array}{l}(max(q^A_{2,2}, q^B_{2,2}),\\ max(r^A_{2,2}, r^B_{2,2}))\end{array} \\ & \cdots\cdots\cdots\cdots & \\ & s^A_1 + s^B_1 & s^A_2 + s^B_2 \end{array}\right)$$

The SPLIT Operator

The SPLIT operator can be used when a node j in the resource graph has more than one outgoing arc. Further, an individual outgoing arc l can be split (creating a separate node) as long as all the jobs traversing the arc in question have node j as their start node (they should not be traversing any incoming arc of node j). Outgoing arcs from node j that do not satisfy this condition cannot be split. The load matrix A of node j is split into two matrices, one for node j and one for the new node j' that is created. The resultant matrix for the new node j' is obtained by replicating matrix A and zeroing out all columns corresponding to jobs that do not traverse arc l, and the matrix for node j is obtained by zeroing out all columns corresponding to jobs that traverse arc l. Further, for any job J_k and a higher priority job J_i, if the two jobs follow different outgoing arcs from node j, the accumulator term of J_k (in the output matrix containing J_k) is updated by replacing the element $(q_{i,k}, r_{i,k})$ with $(0, q_{i,k} + r_{i,k})$. Figure 6.2(b) shows the SPLIT operation, and two hypothetical stages are created after the operation. We formally define the SPLIT operator as follows:

Definition: SPLIT Operator. Let matrix A denote node j and let l be an outgoing arc of node j, such that all jobs X_1 traversing arc l have node j as their start stage. Let X_2 denote the set of jobs that do not traverse arc l. The resultant matrices A_x, $x = 1, 2$, are obtained as follows:

$\forall J_k$:

1. if $J_k \in X_x$: $s_k^{A_x} = s_k^A$; $\forall i$: if $J_i \in X_x$: $q_{i,k}^{A_x} = q_{i,k}^A$, $r_{i,k}^{A_x} = r_{i,k}^A$, else $q_{i,k}^{A_x} = 0$, $r_{i,k}^{A_x} = q_{i,k}^A + r_{i,k}^A$.

2. if $J_k \notin X_x$: $s_k^{A_x} = 0$; $\forall i$: $q_{i,k}^{A_x} = 0$, $r_{i,k}^{A_x} = 0$.

□

The LOOP Operator

Any situation where a PIPE or SPLIT operation cannot be applied to any arc in the graph, implies that a loop exists in the task graph (this follows from the liveness property for the PIPE and SPLIT operations). Consider an outgoing arc l from a node j that is part of a loop. Let X denote the set of jobs that traverse arc l. If the set of jobs that traverse the other outgoing arcs from node j is a subset of X, then the LOOP operator can be applied to arc l. This condition ensures that there is no job whose path is splitting away from the jobs traversing arc l on which the LOOP operator is applied. Like the PIPE operator, the LOOP composes the two nodes A and B at the ends of link l together into a single node. It takes the maximum of corresponding max-terms and the sum of the corresponding accumulator terms in the two operand matrices. If composing the two nodes marks the end of a higher priority task segment, say for task i (all other arcs in the segment have been composed together), then the corresponding resultant max-term $q_{i,k}$ is added to the accumulator $r_{i,k}$, and the max-term is reset to the maximum computation time of J_i on the stage (A or B) from which its next segment starts. Further, if the higher priority task J_i traverses both the forward and the reverse link (that is, traverses the link from A to B as well as the link from B to A), then we add twice the resultant max-term to the accumulator term to account for the interference due to both the forward and reverse flow segments. If a loop is traversed by two tasks J_i and J_k p times (the same sequence of links), then we account for p times the delay component to be added to the accumulator term.

Definition. LOOP Operator. When a PIPE or a SPLIT operation cannot be performed, and node j has an outgoing arc l that is part of a loop, such that the set of all jobs that traverse other outgoing arcs from node j is a subset of the set of jobs that traverse the outgoing arc l from node j, then the LOOP operator can be applied to arc l. Let A and B represent the operand matrices of the nodes that arc l connects, and let C be the resultant operand matrix. $C = A\ LOOP\ B$, is obtained as follows:

1. $\forall i, k$: $q_{i,k}^C = \max(q_{i,k}^A, q_{i,k}^B)$; $r_{i,k}^C = r_{i,k}^A + r_{i,k}^B$

2. $\forall i, k$: If end of higher priority segment of J_i (J_i and J_k traverse loop p times):

   2.1 If J_i traverses both the arc from A to B as well as the arc from B to A, then $r_{i,k}^C = r_{i,k}^C + 2p \times q_{i,k}^C$ else $r_{i,k}^C = r_{i,k}^C + p \times q_{i,k}^C$

2.2 If J_i has outgoing arc from node corresponding to A, then $q^C_{i,k} = q^A_{i,k}$

else if J_i has outgoing arc from node corresponding to B, then $q^C_{i,k} = q^B_{i,k}$

else $q^C_{i,k} = 0$

3. $\forall k$: $s^C_k = s^A_k + s^B_k$

□

The CUT Operator

When the LOOP operator cannot be performed as well, then the CUT operator as defined in Section 4.8 needs to be performed to break a loop in the task graph. Such a situation might arise as the LOOP operator can only be applied to a link l if the set of jobs traversing link l is a superset of the set of jobs traversing other outgoing arcs from the node at the head of link l.

The CUT operation breaks each job traversing the arc being cut into two independent jobs, one for the part before the cut and one for the part after. This operation only relaxes constraints on the arrival times of jobs, allowing jobs to arrive in a manner that can cause worst-case delay (an adversary has greater freedom in choosing the arrival times of jobs to cause a worst-case delay). This decreases the schedulability of the task set and performs a transformation that is safe.

Definition: CUT Operator. When the directed resource graph contains a cycle and when a PIPE, SPLIT, or LOOP operation cannot be performed, a CUT operation can be performed on one of the arcs forming the cycle. Each job crossing that arc is thereby replaced by two independent jobs; one for the part before the cut and one for the part remaining. Each new job will have a separate row and column in the operand matrices for stages on which they execute. □

Observe that, for a system with n jobs, every operand matrix has $n + 1$ rows and n columns. Any job that does not execute at a stage has a column with all its elements set to zero. It is possible to optimize this representation by removing all zero-element columns and having operand matrices of variable dimensions. Row and column indices would have to be represented explicitly (rather than the implicit global job-numbering assumed in the above exposition).

The definition of the operators are the same regardless of whether the scheduling is preemptive or non-preemptive. By successively applying the operators of the algebra, the distributed system can be reduced to a single equivalent uniprocessor. Note that as the max and sum operations are commutative and associative, the PIPE and LOOP operators are commutative and associative as well.

Proof of Liveness and Algorithm Complexity

Given the operator definitions described above, we now prove the following theorem:

Theorem: *The delay composition algebra always reduces the original resource graph (augmented with the extra finish node) to a single node.*

Proof: For simplicity, we prove the theorem only for the case of Directed Acyclic Graphs, using the PIPE and SPLIT operators. The proof of the theorem for the LOOP and CUT operators included follows trivially. To prove the theorem, observe that we defined the following rules for applying the algebraic operators: (i) a PIPE can only be applied to a pair of nodes if the upstream node has exactly one outgoing arc, and (ii) a SPLIT can only be applied to a node if it has no incoming arcs and multiple outgoing arcs.

Hence, a PIPE can always be performed unless we are left only with those nodes that have *multiple* outgoing arcs (and their immediate downstream neighbors). However, in such a case, a SPLIT can always be performed on the earliest of these nodes. This is because (i) this node does not have incoming arcs from earlier nodes (that would contradict it being earliest), and (ii) it has multiple outgoing arcs (since only such nodes are left together with their downstream neighbors but the earliest node, by definition, is not downstream from another). Hence, at any given time, either a PIPE or a SPLIT can always be performed until no arcs are left.

It is left to show that the graph always remains connected, and hence when no arcs are left only one node remains. We prove it by induction. First note that the initial DAG is connected, and the virtual finish node f is downstream from every node. This is because each node j is either an end node of some job, in which case it is connected directly downstream to the virtual finish node, f, or is not an end node, in which case it must have a downstream path to the end node of some job, and the latter is connected downstream to the virtual finish node. Hence, the finish node can be reached from any node by a downstream path and the graph is connected. Next, we prove the induction step, showing that applying a PIPE or SPLIT does not disconnect the graph and keeps f downstream from every node. For a PIPE, this is self-evident, since it only merges nodes. For a SPLIT, assume that the graph before the SPLIT was applied was connected and each node had a downstream path to the virtual finish node. SPLIT takes a node j with an immediate downstream neighbor set N_j and replaces it with multiple nodes, each inheriting a downstream arc to one of these neighbors. Thus, since neighbors in set N_j are connected to the virtual finish node by a downstream path, so will be each of the newly created nodes. The induction hypothesis is maintained. By induction, the graph is never disconnected by PIPEs or SPLITs, and the finish node is always downstream from every node. Hence, when the algebra has removed all arcs, the DAG is reduced to a single node. □

A PIPE operation can be performed in $O(n^2)$ time, where n is the number of jobs in the system, and each PIPE operation reduces the number of arcs in the graph by one. The time complexity for a SPLIT operation involving k arcs is $O(kn^2)$, and each of these k arcs can be eliminated through PIPE operations

in the next step. Hence, the complexity for eliminating each arc is $O(n^2)$, and the net complexity of the algebra to reduce a graph to a single node is $O(|E|n^2)$, where $|E|$ is the number of arcs in the original resource graph.

6.1.4 Task Set Transformation

Once the system is reduced to one node, the end-to-end delay and schedulability of any job J_k can be inferred from the node's load matrix. Remember that in periodic or sporadic task systems, J_k stands for an instance of task T_k. We shall use the task notation in this section, since we expect the algebra to be applied mostly for periodic or sporadic task sets. Once the system is reduced to one node, the max-term for each element in the final matrix is first added to the accumulator term, that is, $(q_{i,k}, r_{i,k})$ is replaced by $(0, q_{i,k} + r_{i,k})$. To analyze the schedulability of any task T_k in the original distributed system, an equivalent uniprocessor task set is obtained from column k of the final load matrix as follows:

- Each task T_i, $i \neq k$ in the original distributed system is transformed to task T_i^* on a uniprocessor, with a computation time $C_i^* = r_{i,k}$, if scheduling is non-preemptive, or $C_i^* = 2r_{i,k}$, if scheduling is preemptive (the reason for which is explained in Section 6.2). The period P_i (if T_i is periodic) or minimum inter-arrival time (if it is sporadic) remains the same (i.e., $P_i^* = P_i$).
- Task T_k, for which schedulability analysis is performed, is transformed to task T_k^* with $C_k^* = r_{k,k}$ plus an extra task of computation time s_k. The period or minimum inter-arrival time for both, remains that of T_k.

We prove in Section 6.2 that if T_k^* meets its deadline on the uniprocessor when scheduled together with this task set, then T_k meets its deadline in the original distributed system. Any uniprocessor schedulability test can be used to analyze the schedulability of T_k^*. Note that a separate test is needed per task. First, however, we present an example.

6.1.5 An Illustrative Example

We now illustrate how the algebra can be applied to a distributed system to reduce it to a single equivalent hypothetical uniprocessor for the purpose of analyzing the end-to-end delay and schedulability of jobs in the original distributed system. We consider a system of four resource stages shown in Figure 6.3(a), and three periodic tasks T_1, T_2, and T_3, in decreasing priority order. T_1 follows the path $S_1 - S_2 - S_3 - S_1 - S_4$, T_2 follows $S_1 - S_2 - S_3 - S_4$, and T_3 follows $S_1 - S_2 - S_4$. Each task invocation requires one unit of computation time at each resource along its path, and the relative end-to-end deadline is assumed to be the same as the task period. T_1 has a period of 10 units, and T_2 and T_3 have a period of 20 units. We do not need to create a virtual finish node as all task routes end at the same finish node (S_4).

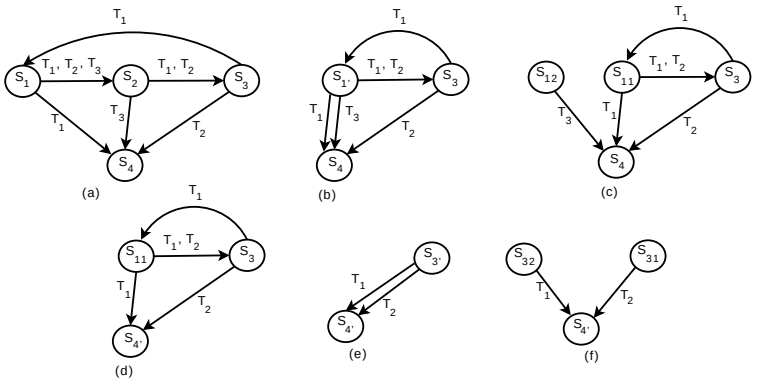

Figure 6.3: (a) Example system to be composed (b) Composed system after step 1 (c) Composed system after step 2 (d) After step 3 (e) After step 4 (f) After step 5

Let A_i denote the operand matrix for stage S_i. The initial operand matrices are constructed as shown below. As task T_1 executes twice on stage S_1, the stage-additive component s_1 of A_1 is two, while all other stage-additive component values are one.

$$A_1 = \left(\begin{array}{c|ccc} & T_1 & T_2 & T_3 \\ \hline T_1 & (1,0) & (1,0) & (1,0) \\ T_2 & (0,0) & (1,0) & (1,0) \\ T_3 & (0,0) & (0,0) & (1,0) \\ & \cdots & \cdots & \cdots \\ & 2 & 1 & 1 \end{array}\right), A_2 = A_4 = \left(\begin{array}{c|ccc} & T_1 & T_2 & T_3 \\ \hline T_1 & (1,0) & (1,0) & (1,0) \\ T_2 & (0,0) & (1,0) & (1,0) \\ T_3 & (0,0) & (0,0) & (1,0) \\ & \cdots & \cdots & \cdots \\ & 1 & 1 & 1 \end{array}\right), A_3 = \left(\begin{array}{c|cc} & T_1 & T_2 \\ \hline T_1 & (1,0) & (1,0) \\ T_2 & (0,0) & (1,0) \\ & \cdots & \cdots \\ & 1 & 1 \end{array}\right)$$

All nodes have 2 out-going arcs, and no PIPE or SPLIT operations can be performed. A loop exists, and we apply the LOOP operator to arc $S_1 - S_2$.

Step 1: $A_1\ LOOP\ A_2\ =\ A_{1'}$

We take the maximum of the corresponding max-terms and the sum of the corresponding accumulator terms and the stage-additive components. The arc under consideration does not mark the end of T_1's segment when considering the delay of T_2. But, it marks the end of the segment of T_1 that interferes with T_3. As T_3 executes on only one of the arcs and does not traverse an arc from S_2 to S_1, it contributes only one unit of delay, which is added to the accumulator term.

The resultant task graph is as shown in Figure 6.3(b). Now, stage $S_{1'}$ is the start stage for T_3, and T_3 is the only job that traverses the arc from $S_{1'}$ to S_4 (note that T_1 traverses a different arc from $S_{1'}$ to S_4). We can therefore apply the SPLIT operator to split T_3 along that arc creating nodes S_{11} and S_{12}, whose operand matrices are as follows:

Step 2: $SPLIT(S_{1'}, \{T_3\})\ =>\ A_{11}, A_{12}$

$$A_{1'} = \left(\begin{array}{c|ccc} & T_1 & T_2 & T_3 \\ \hline T_1 & (1,0) & (1,0) & (1,1) \\ T_2 & (0,0) & (1,0) & (1,0) \\ T_3 & (0,0) & (0,0) & (1,0) \\ & \cdots & \cdots & \cdots \\ & 3 & 2 & 2 \end{array}\right), A_{11} = \left(\begin{array}{c|cc} & T_1 & T_2 \\ \hline T_1 & (1,0) & (1,0) \\ T_2 & (0,0) & (1,0) \\ T_3 & (0,0) & (0,0) \\ & \cdots & \cdots \\ & 3 & 2 \end{array}\right), A_{12} = \left(\begin{array}{c|c} & T_3 \\ \hline T_1 & (0,2) \\ T_2 & (0,1) \\ T_3 & (1,0) \\ & \cdots \\ & 2 \end{array}\right)$$

Figure 6.3(c) shows the updated task graph. S_{12} can now be piped with S_4 to give $S_{4'}$.

Step 3: $A_{12}\ PIPE\ A_4\ =\ A_{4'}$

The resultant task graph is shown in Figure 6.3(d). Again nodes have more than one out-going arc and no PIPE or SPLIT operations can be performed. We perform a LOOP operation on the arc $S_{11} - S_3$ to merge the nodes into a single node $S_{3'}$. This operation marks the end of the task segment of T_1 that delays T_2, and T_1 traverses the arc from S_{11} to S_3, as well as the arc from S_3 to S_{11}. T_1 can delay T_2 both in the forward as well as reverse directions, and we need to account for two units of delay, which is added to the accumulator term.

Step 4: $A_{11}\ LOOP\ A_3\ =\ A_{3'}$

This leaves us with two nodes $S_{3'}$ and $S_{4'}$ with two arcs connecting them, one traversed by T_1 and the other by T_2, as shown in Figure 6.3(e). We can now split node $S_{3'}$ into two nodes S_{31} and S_{32} one for each of the out-going arcs from $S_{3'}$.

Step 5: $SPLIT(S_{3'}, \{T_1\})\ =>\ A_{31}, A_{32}$

$$A_{4'} = \left(\begin{array}{c|ccc} & T_1 & T_2 & T_3 \\ \hline T_1 & (1,0) & (1,0) & (1,2) \\ T_2 & (0,0) & (1,0) & (1,1) \\ T_3 & (0,0) & (0,0) & (1,0) \\ & \cdots & \cdots & \cdots \\ & 1 & 1 & 3 \end{array}\right), A_{3'} = \left(\begin{array}{c|cc} & T_1 & T_2 \\ \hline T_1 & (1,0) & (1,2) \\ T_2 & (0,0) & (1,0) \\ T_3 & (0,0) & (0,0) \\ & \cdots & \cdots \\ & 4 & 3 \end{array}\right), A_{31} = \left(\begin{array}{c|c} & T_1 \\ \hline T_1 & (1,0) \\ T_2 & (0,0) \\ T_3 & (0,0) \\ & \cdots \\ & 4 \end{array}\right), A_{32} = \left(\begin{array}{c|c} & T_2 \\ \hline T_1 & (0,3) \\ T_2 & (1,0) \\ T_3 & (0,0) \\ & \cdots \\ & 3 \end{array}\right)$$

This leaves us with the task graph shown in Figure 6.3(f). We can now independently PIPE S_{31} and S_{32} with $S_{4'}$, to get S_{final}.

Step 6: $A_{31}\ PIPE\ A_{4'}\ =\ A_{4''}$, $A_{32}\ PIPE\ A_{4''}\ =\ A_{final}$

$$A_{final} = \left(\begin{array}{c|ccc} & T_1 & T_2 & T_3 \\ \hline T_1 & (1,0) & (1,3) & (1,2) \\ T_2 & (0,0) & (1,0) & (1,1) \\ T_3 & (0,0) & (0,0) & (1,0) \\ & \cdots\cdots\cdots\cdots\cdots & & \\ & 5 & 4 & 3 \end{array}\right) = \left(\begin{array}{c|ccc} & T_1 & T_2 & T_3 \\ \hline T_1 & (0,1) & (0,4) & (0,3) \\ T_2 & (0,0) & (0,1) & (0,2) \\ T_3 & (0,0) & (0,0) & (0,1) \\ & \cdots\cdots\cdots\cdots\cdots & & \\ & 5 & 4 & 3 \end{array}\right)$$

With this final equivalent single stage matrix, we can construct a uniprocessor task set and use any uniprocessor schedulability test to analyze the schedulability of a task in the distributed system. The reduction process and schedulability analysis is similar to the description in [39], and is omitted in the interest of brevity.

6.2 Proof of Correctness

In this section, we prove the correctness of the delay composition algebra. By correctness, we mean that if a job is schedulable in the resulting uniprocessor task set, it is schedulable in the original distributed system. Below, we show the proof for preemptive systems. The proof for non-preemptive systems is similar and is thus omitted. Consider a job J_k that executes along a path p_k in the original directed acyclic graph. It is desired to determine the schedulability of J_k. Consider a higher-priority job J_i ($i \neq k$) that executes along path p_i. Let paths p_k and p_i intersect in some set $Seg_{i,k}$ of sequences of consecutive (i.e., directly connected) nodes. For example if J_k has the path (1, 2, 5, 8, 11, 13) and J_i has the path (9, 1, 2, 16, 8, 11, 10) then $Seg_{i,k} = \{(1,2),(8,11)\}$. Each member of this set is a shared path segment between J_k and J_i. Let the part of J_i that executes on segment s in set $Seg_{i,k}$ be called sub-job J_i^s. In the above example, J_i^1 is the part of J_i that executes on the path segment (1, 2) and J_i^2 is the part of J_i that executes on the path segment (8, 11). Note that sub-jobs J_i^s are the only parts of J_i that may delay J_k since they are the only parts that share (part of) J_k's path. Let the maximum execution time of sub-job J_i^s over its path be $C_{i,max}^s$. Let the maximum execution time of all jobs J_i^s on node j be $Node_{j,max}$. The delay composition theorem applied to a job J_k and the set S of higher priority job-segments J_i^s that share a sequence of consecutive common execution stages with J_k is as follows:

$$Delay(J_k) \leq \sum_i \sum_{J_i^s \in S} 2C_{i,max}^s + \sum_{j \in p_k} Node_{j,max} \tag{6.1}$$

The above inequality can be rewritten as follows:

$$Delay(J_k) \leq \sum_i 2r^*_{i,k} + s^*_k \tag{6.2}$$

$$r^*_{i,k} = \sum_{J^s_i \in S} C^s_{i,max}; \quad s^*_k = \sum_{j \in p_k} Node_{j,max} \tag{6.3}$$

Let $r^M_{i,k}$ denote the $(i,k)^{th}$ element in the final single stage matrix derived using the algebra. Since delays due to higher priority jobs are additive on a uniprocessor, the delay that the transformed job J_k, called J^*_k, experiences on the hypothetical uniprocessor is precisely $Delay(J^*_k) = \sum_i 2r^M_{i,k} + s^M_k$ (after multiplying $r^M_{i,k}$ by 2 as per rules in Section 6.1.4). If $r^M_{i,k} = r^*_{i,k}$ and $s^M_k = s^*_k$, it follows that $Delay(J_k) \leq Delay(J^*_k)$. Thus, if J^*_k is schedulable on the uniprocessor, so is J_k in the original distributed system. In the case of periodic tasks, as observed in [39], finding the actual number of invocations, $Invoc_i$, for each higher priority periodic task, T_i, that delays J_k, is not the responsibility of the algebra or the reduction process. This is handled by the uniprocessor analysis used. The number of invocations, $Invoc_i$, as determined by the uniprocessor analysis will at least be as large as the number of actual invocations of T_i that delay J_k in the distributed system. The reason is because, every invocation of T^*_i that arrives before J^*_k completes execution will delay J^*_k on the uniprocessor, but the corresponding invocations of T_i may never catch-up with J_k to preempt it in the distributed system, as they may be executing on different resources.

We shall now show that in the final matrix, $r^M_{i,k} = r^*_{i,k}$ and $s^M_k = s^*_k$. It is safe to assume that all necessary CUT operations are performed first, as any CUT operation only relaxes precedence constraints and performs a safe transformation of the system that does not improve schedulability. Now, consider the entire sequence of PIPE, SPLIT, and LOOP operations performed to reduce the distributed system to a single node. Let us denote each arc using an unique identifier, and let the set of all arcs in the original distributed system be denoted by L^0. Note that SPLIT operations neither add nor remove arcs from L^0. PIPE operations remove precisely one arc from L^0, and LOOP operations may remove at most two arcs (connecting the same two nodes) from L^0.

In order to compute $r^M_{i,k}$, consider the path of J_k. Let L^0_k denote the subset of arcs in L^0 that lie on the path of J_k (including arcs in the opposite direction as J_k). As in the proof presented in [39], all PIPE operations can be classified under three categories: *path* PIPEs (applied to an arc in L^0_k), *incident* PIPEs (applied to an arc that shares one node with an arc in L^0_k), and *detached* PIPEs (applied to an arc that shares no nodes with arc in L^0_k). SPLIT operations can be classified into two categories: *path* SPLITs (applied to a node with an arc in L^0_k) and *detached* SPLITs (the rest). Likewise, LOOP operations can be classified into *path*, *incident*, and *detached* LOOPs. Trivially, only path PIPEs, path SPLITs, and path LOOPs affect

elements in column k of the operand matrices, that is, the components of the delay of job J_k.

Consider a job J_i of higher priority than J_k. Let us denote the set of arcs in $Seg_{i,k}$ (those arcs traveled by both J_i and J_k as $L^0_{i,k}$. Path PIPEs and path LOOPs that reduce arcs not traveled by J_i simply propagate $q_{i,k}$ of the downstream node, and the sum of the $r_{i,k}$'s of the upstream and downstream nodes to the resultant matrix. This is because, as J_i does not travel the reduced arc it does not execute on the upstream node and $q_{i,k}$ of the upstream node must be zero. The $r_{i,k}$ values in the upstream and downstream nodes denote the delay of independent job-segments of J_i which need to be added together. Further, SPLITs of nodes with no arcs traveled by J_i do not alter $q_{i,k}$ and $r_{i,k}$, since J_i could not have parted J_k at the split node. Hence, we now need to only consider PIPE, LOOP, and SPLIT operations involving arcs traveled by both jobs (that is, in $Seg_{i,k}$).

Consider a segment $s \in Seg_{i,k}$. Let E_s be the last node in the segment. Initially, each node $j \in s$ has $q_{i,k}$ set to the maximum computation time of J_i over all its visits to stage j. To reduce each arc in the segment, a PIPE or LOOP operation must have been performed, producing $q_{i,k}$ to be equal to the maximum of all the computation times of J_i over all the stages in the segment. Recall that a LOOP operation is performed on an arc only when the set of jobs that traverse the arc is a super set of the set of jobs that traverse other outgoing arcs from the node. This ensures that there are no jobs that split from the path of jobs following the arc on which the LOOP operation is performed. Further, note that for each stage, by taking $q_{i,k}$ to be the maximum computation time of J_i over all its visits to the stage, we overestimate the delay of J_k as compared to the delay composition theorem. This is essential, as there is no information stored in the operand matrix with regard to which visit of J_i corresponds to the current segment, and it is safe to consider the maximum computation time over all the visits to the stage. Any SPLIT operation performed on nodes $j \in s$, other than the last node and involving J_i, does not affect $q_{i,k}$ or $r_{i,k}$, as J_k and J_i did not part ways at stage j. Only LOOP and SPLIT operations involving E_s affect the value of $r_{i,k}$.

A LOOP operation that involves E_s and marks the end of the segment s (i.e., removes the last arc that is part of segment s from the task graph), causes $r_{i,k}$ to be updated by adding $q_{i,k}$ to it, which by now equal to the maximum of all the computation times of J_i over all the stages in the segment. If J_i traverses both the forward and reverse arc between the two nodes, then J_i could potentially delay J_k twice, and twice the value of $q_{i,k}$ needs to be added to $r_{i,k}$. If J_i loops back and a new segment begins at one of the two nodes involved, then $q_{i,k}$ is reset to denote the maximum computation time of J_i on that node. At node E_s, any SPLIT operation that splits J_i from J_k can be performed only when there is no incoming arc into E_s that is traversed by J_i. This implies that all PIPE and LOOP operations have been applied over all the other nodes in the segment. Hence, at E_s, $q_{i,k} \geq C^s_{i,max}$ ($q_{i,k}$ may be larger than $C^s_{i,max}$ as the maximum

computation time of J_i over all visits on all stages is taken for each stage operand matrix). The SPLIT would then add $q_{i,k}$ to $r_{i,k}$. As noted before, subsequent operations propagate $r_{i,k}$ to the result node. When all the segments have been reduced, $C^s_{i,max}$ for each segment s is added on to $r_{i,k}$, resulting in $\sum C^s_{i,max}$ over all job-segments J^s_i. In other words, $r^M_{i,k} = r^*_{i,k}$. (Observe that, if J_k and J_i have the same end node, there would be no SPLIT for the last segment and its max-term would still be stored in $q_{i,k}$; this is why we need to compensate for the missing SPLIT and manually add $q^M_{i,k}$ and $r^M_{i,k}$ at the end.)

Similarly, to compute s^M_k, observe that initially for each node j on path p_k, $s^j_k = Node_{j,max}$. Since SPLITs do not affect s_k and PIPEs and LOOPs add it, when all arcs on L^0_k are reduced, $s^M_k = \sum_{j \in p_k} Node_{j,max} = s^*_k$. □

6.3 Evaluation

In this section, we evaluate the accuracy and tightness of the delay composition algebra in estimating the end-to-end delay and schedulability of jobs, for the case of directed acyclic graphs. Our custom-built simulator that models periodic tasks executing in a distributed acyclic system is used. An admission controller based on the delay composition algebra is used to guarantee the deadlines of tasks in the system. The analysis is meant as a design-time capacity-planning tool and hence the need for global knowledge by the admission controller is not a problem.

We consider two measures of performance. First, we estimate the average ratio of the end-to-end delay of tasks to their computed worst-case end-to-end delay bound. This metric shows how pessimistic the theoretically computed worst-case is (as per each approach) compared to the average case. Second, we consider the average per-stage utilization of tasks admitted into the system and is a measure of the throughput of the system. Utilization of a resource is defined as the fraction of time the resource is busy servicing a task. We compare our analysis using the delay composition algebra with holistic analysis [89] and network calculus [18, 19], under both preemptive and non-preemptive scheduling. We use the result from [52], for holistic analysis under non-preemptive scheduling. We build an admission controller for each analysis technique (delay composition algebra, holistic analysis, and network calculus) and compare the conservatism of the various analyses with respect to admission control.

Simulation parameters are chosen similar to the evaluation in previous chapters. The default value of the Node Probability parameter, NP, is 0.8. The default value for the Deadline Ratio parameter, DR, is 2.0. The default value for the task resolution parameter, T, is 1/20.

First, we ascertain that the performance does not significantly drop with increasing system size. We measured the average ratio of end-to-end delay of jobs to the calculated upper bound on the worst-case

delay, as a function of system size. The results are shown in Figure 6.4.

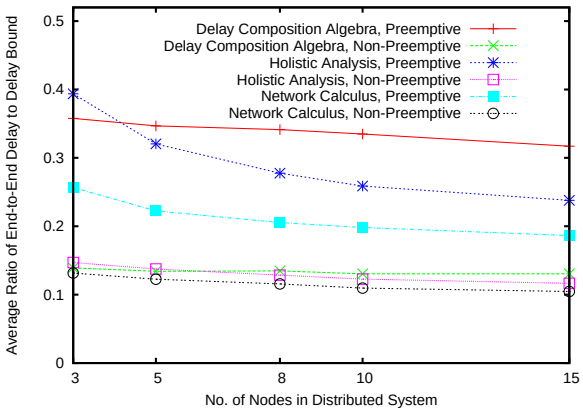

Figure 6.4: Comparison of average ratio of end-to-end delay to estimated delay bound for different number of nodes in the system

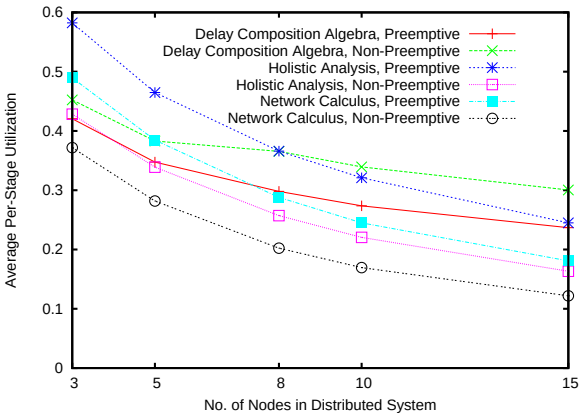

Figure 6.5: Comparison of average per-stage utilization for different number of nodes in the system

For the delay composition algebra, under both preemptive and non-preemptive scheduling, the ratio remains nearly the same regardless of system size, showing that the pessimism in analysis does not increase with system scale. However, holistic analysis tends to be increasingly pessimistic with system scale, and the ratio drops with increasing number of nodes in the system. The ratio is lower for non-preemptive scheduling, as there are several low priority jobs that finish well before their worst-case delay estimate as they are not preempted by higher priority jobs and therefore encounter only a smaller fraction of all higher priority jobs during their execution (on an average) than under preemptive scheduling.

For the same experiment, Figure 6.5 plots the average per-stage utilization of admitted tasks. Note that, the drop in average utilization is faster for holistic analysis and network calculus than for our algebraic analysis with increasing system size. Holistic analysis consistently outperforms network calculus for all system sizes.

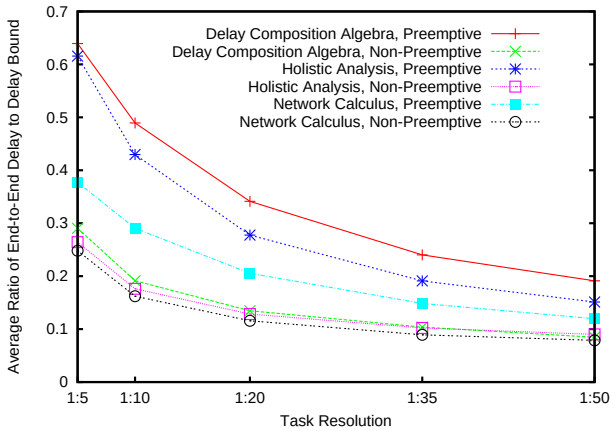

Figure 6.6: Comparison of average ratio of end-to-end delay to estimated delay bound for different task resolution values

α	Compositional Analysis		Holistic Analysis	
	NP	P	NP	P
1	0	0	0	0
2	0.003	0	0.009	0
3	0.018	0	0.037	0
5	0.072	0	0.108	0
10	0.151	0	0.145	3×10^{-6}
20	0.150	0	0.151	3.3×10^{-5}
50	0.150	6×10^{-6}	0.152	1.56×10^{-4}

Table 6.1: Fraction of deadlines missed for different values of the deadline scaling factor α

We next varied the size of jobs by adjusting the task resolution parameter T. A large value for T (e.g., 1:5) denotes a system with a small number of large tasks, and a small value of T (e.g., 1:50) denotes a large number of small tasks. We measured the ratio of the end-to-end delay to the delay bound for the three analysis techniques under both preemptive and non-preemptive scheduling, and the results are shown in Figure 6.6. Delay composition algebra tends to be the least pessimistic under preemptive as well as non-preemptive scheduling. As the number of tasks in the system increases (as T decreases), jobs encounter a smaller fraction of higher priority jobs, and therefore the average end-to-end delay significantly differs from the worst-case delay. Under non-preemptive scheduling, when task sizes are large (T is large) the blocking penalty for higher priority jobs is also high, although on an average jobs are not blocked for the estimated worst-case period. This causes the ratio under non-preemptive scheduling to be lower than under preemptive scheduling.

In the previous experiments, we measured the average ratio of the end-to-end delay to the estimated worst-case delay bound. However, this provides no insight into the variance. In other words, are there jobs whose delay is very close to their worst-case delay, while other jobs finish well ahead of their delay bound? To answer this question, we scaled the deadlines of tasks in the admission controller by a factor $\alpha \geq 1$. Thus, the admission controller would admit more tasks than deemed feasible by the analysis, and we measured the fraction of deadlines that were missed. For different values of α, Table 6.3 presents the average fraction of deadlines missed for the compositional analysis as well as holistic analysis.

Under non-preemptive scheduling, it is observed that deadlines are missed more frequently and for smaller values of α, than under preemptive scheduling. The reason for the more frequent deadline misses is that higher priority jobs have a much higher ratio of average delay to worst-case delay than lower priority jobs (not shown in the results), under non-preemptive scheduling. So, when the deadline values are scaled in the admission controller, the higher priority jobs immediately miss their deadlines. This is confirmed by the fact that the fraction of deadlines missed saturates at about 15% when α is increased beyond 10, as the lower priority jobs have a very low average delay and do not miss their deadlines even if the deadline values are

scaled up in the admission controller. On the other hand, the variance in the ratio of average delay to the worst-case delay bound is much lower under preemptive scheduling. Therefore, although deadlines are scaled by up to a factor of 20, admitting several more tasks, no deadline misses are observed (in the average case). From Figure 6.4, the end-to-end delay bound is about 3 times the average delay of jobs under preemptive scheduling (ratio value of 0.35). An alternate way of interpreting the results in Table 6.3 is that, under preemptive scheduling, increasing the average delay by a factor of 3, increases the worst-case delay by a factor of 20-50. This suggests that for systems where worst-case delay is critical, non-preemptive scheduling is perhaps a better choice. In contrast, for systems where only the average delay is of interest, preemptive scheduling would work well.

Chapter 7

Flow-based Mode Changes: Virtual Uniprocessor Models for Reduction-based Analysis

In this chapter, we develop a new task model for uniprocessors, motivated by the needs of reduction-based schedulability analysis techniques for distributed real-time systems. The resulting uniprocessor workloads constructed by reducing distributed system workloads are subject to additional constraints not previously considered in uniprocessor literature. The current practice has been to ignore these constraints (on worst-case load), resulting in needlessly pessimistic worst cases, and hence in pessimistic schedulability estimates for workloads reduced from distributed systems. The problem motivates new uniprocessor workload models that serve the needs of reduction-based schedulability analysis literature.

The fundamental idea of workload transformation in reduction-based schedulability analysis is to show that when two periodic distributed tasks execute together on a sequence of machines (called *stages*) in a distributed system, each invocation of the higher-priority task delays an invocation of the lower-priority task by a *bounded* total amount in the entire system. This bounded amount is computed and becomes the transformed execution time of an equivalent periodic higher-priority task on a virtual uniprocessor. Analyzing the schedulability of the lower priority task subject to all such transformed higher-priority uniprocessor periodic tasks then determines the schedulability of the former in the original distributed system.

The source of pessimism arises due to the fact that once all tasks have been reduced to a single periodic uniprocessor task set, uniprocessor analysis assumes that these tasks execute together "all the time", whereas in fact they may share a machine only for *part* of their execution in the original distributed system. Hence, the number of interfering invocations of higher-priority tasks may be overestimated.

We address the pessimism in current reduction-based schedulability analysis techniques by introducing *flow-based mode changes*, a novel model for uniprocessor workloads featuring mode changes that mimic what happens when a distributed task moves from one processor to another in a distributed system. A key distinction of our model as compared to the classical literature on mode changes, such as [81, 88, 78, 74], is that we do not precisely know when the mode changes will occur, as we do not know when exactly the task changes machines. However, we have constraints on mode change timing that stem from bounds on task delays on different machines. Therefore, the problem is one of estimating the response time of a task for the

worst-case scenario of mode changes subject to new mode change constraints.

We present an iterative solution to solve the above problem, providing significantly tighter estimates on the number of higher priority task invocations that delay the task under consideration. The solution converges to a delay bound that never underestimates the worst-case delay of the corresponding task in the distributed system. Our simulations demonstrate that the presented analysis provides an improvement in performance of over 25% compared to existing techniques, in terms of admissible utilization.

The rest of the chapter is organized as follows. In Section 7.1, we describe the new multi-modal uniprocessor system model proposed. We present an iterative solution to determine the response time of a task under consideration in such a system in Section 7.2. In Section 7.3, we show the reduction of a distributed system to such a multi-modal uniprocessor system for the purpose of schedulability analysis. We also present an example to illustrate the advantage of the proposed solution over previous reduction based analysis techniques for distributed systems. We evaluate the technique through simulation in Section 7.4.

7.1 Multi-Modal Uniprocessor System Model

In this section, we present the new multi-modal uniprocessor system model of interest in this work. This model is motivated by the needs of reduction-based schedulability analysis in distributed systems. It is thus important to first highlight the relation between distributed task execution models and the model below.

Reduction-based schedulability analysis addresses scenarios where tasks execute on a sequence of machines in a distributed system and must each finish within its end-to-end deadline. Consider a distributed system, running fixed priority scheduling, where tasks T_1, $T_2, \ldots, T_m$, are executed, indexed in decreasing priority order. Task T_i executes on a sequence of n_i machines. It therefore comprises of a sequence of n_i jobs $T_{i,1}, T_{i,2}, \ldots, T_{i,n_i}$, each running on the corresponding machine. Job $T_{i,j+1}$ becomes ready to execute as soon as $T_{i,j}$ completes execution, at which point task T_i is said to have switched to the next machine on its path. The model fits a pipelined execution scenario, where a task is broken up into a number of sequential stages that must execute in some given order.

It is now possible to plot the execution of task T_i from its entry to the system to its exit from the system on a single time line. This time line will comprise one busy period (i.e., a period with no idle time) composed of intervals when the task was delayed or preempted by others on some machine as well as intervals where it was executing. The finish time of each job $T_{i,j}$ in that time line corresponds to a time when T_i switches machines and starts competing with a different task set. Assume we are able to accurately bound the delay that each invocation of a higher priority task T_j, $j < i$, imposes on T_i in that time line (which is what reduction-based schedulability analysis literature does). One will then need only to bound the maximum

number of invocations of each T_j that may delay T_i in order to bound T_i's end-to-end response time. This is not straightforward, however, because T_i competes with potentially different subsets of higher priority tasks on each machine, and the exact times it changes machines are not known accurately as they depend on the relative timing of task arrivals. To bound the delay of T_i, schedulability analysis of this time line can then benefit from a uniprocessor model where "mode changes" occur at $T_{i,j}$'s completion times. The objective of that model is to come up with the worst-case timing for "more changes" that maximize T_i's response time (e.g., letting T_i spend longer on machines with heavier load).

Note that, the above maximization problem does not simply amount to the sum of T_i's worst-case response times on each machine. This would be needlessly pessimistic. For example, if T_i and the higher-priority tasks followed the same path, arriving at the first machine together (a worst-case arrival scenario on the first machine), then they will arrive to the next machine staggered by their execution times on the first (which is not a worst-case arrival scenario). Below, we present a multi-modal uniprocessor model that achieves a much tighter response-time bound, and then describe how it can be used for analysis of distributed workloads.

Consider a uniprocessor that uses fixed priority preemptive scheduling. We consider a set of m tasks $T_1^*, T_2^*, \ldots, T_m^*$, in decreasing priority order. Task T_m^*, the lowest priority task whose worst-case response time we wish to estimate, comprises of a sequence of n jobs $T_{m,1}^*, T_{m,2}^*, \ldots, T_{m,n}^*$, with computation times $C_{m,1}^*, C_{m,2}^*, \ldots, C_{m,n}^*$, respectively. The jobs are such that $T_{m,j+1}^*$ is ready to execute as soon as $T_{m,j}^*$ completes execution, for $1 \leq j \leq n-1$. Job $T_{m,1}^*$ is ready to execute at time zero. The time instant of completion of each of the jobs $T_{m,j}^*$ denotes a mode change in the system, where one of the other $m-1$ tasks may arrive or leave the system. Tasks T_i^*, $i \leq m-1$ are periodic tasks with period P_i and computation time C_i^*. Each task T_i^* arrives at the system at either time zero, or during one of the mode changes in the system (time instants of completion of $T_{m,j}^*$, for some $j < n$), and leaves the system at one of the mode changes or when $T_{m,n}^*$ completes execution. Thus, each periodic task executes during some consecutive subset of modes in the system and does not undergo any change within this subset of modes until it leaves the system. The subset of modes in which a task executes is assumed to be known. Hence, during each mode $mode_j$, $j \leq n$ of execution, some pre-defined subset of periodic tasks L_j is present in the system.

We assume that task preemptions and mode changes are instantaneous. We also assume that the cumulative utilization of all the tasks executing during any mode is at most 100%. The objective is to estimate a worst-case bound on the response time of $T_{m,n}^*$ starting from time zero, over all possible scenarios of mode changes. Note that, unlike traditional multi-modal analysis, we are not interested in the schedulability of all tasks in the system, but are interested only in that of $T_{m,n}^*$. As each task in the distributed system could have a different worst-case scenario, the analysis is conducted one task at a time and a different multi-modal

uniprocessor system is constructed and analyzed each time. We later show in Section 7.3, how a distributed task set can be reduced to such a multi-modal uniprocessor, and how the computed bound for $T^*_{m,n}$ also bounds the end-to-end delay of the corresponding task in the distributed system.

7.2 Schedulability Analysis

In Section 7.2.1, we present an algorithm to determine the worst-case response time of T^*_m in the multi-modal uniprocessor system (we use T^*_m and $T^*_{m,n}$ interchangeably to denote the entire task invocation or its last stage when analyzing completion time). We illustrate the algorithm using an example in Section 7.2.2. We comment on the time complexity of the algorithm in Section 7.2.3.

7.2.1 Algorithm Description

Consider Figure 7.1 that demonstrates the execution of task T^*_m, the (yet to be determined) instances of mode changes, and the arrival and departure of higher priority tasks. The completion of the sub-task $T^*_{m,j}$ denotes the completion of $mode_j$, for each j. Let the set of tasks that execute in $mode_j$ be denoted by L_j.

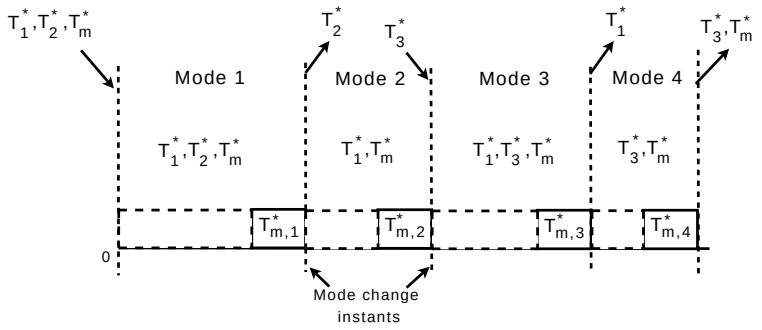

Figure 7.1: Example demonstrating the instants of mode changes and the arrival and departure of higher priority tasks

Let $RT(s, s+q)$, $s \geq 0, q \geq 1$, denote the maximum possible duration between the completion of $mode_s$ and the completion of $mode_{s+q}$. For notational simplicity, we consider time zero (the arrival of task T^*_m) to be the "completion" of a fictional $mode_0$. Therefore, we are interested in computing $RT(0, n)$, which denotes the worst-case response time of $T^*_{m,n}$.

Given the set of tasks L_s that execute at each mode $mode_s$, we can apply response time analysis [8] to compute the maximum response time, $RT(s, s+1)$ for each mode taken independently. Adding up these worst-case single-mode response times for $s = 0, ..., n-1$ would give us an upper bound on $RT(0, n)$. However, such an upper bound will be unduly pessimistic. To appreciate the reason for pessimism, consider a task T^*_i that executes in modes $mode_s$ and $mode_{s+1}$ (e.g., task T^*_3 in Figure 7.1 that executes in $mode_3$ and $mode_4$). Let the period of T^*_i be larger than the total length of the two modes combined (remember

that the end of each $mode_s$ is defined as the instant when $T^*_{m,s}$ ends, and hence is not necessarily aligned with periods of other tasks). Since there can only be one invocation of T^*_i in each period, this invocation will execute either in $mode_s$ or $mode_{s+1}$ but not both. The worst-case response time computed for each mode separately will have to account for this invocation of T^*_i. However, adding these worst-case response times will erroneously double-count this invocation. Hence, in general:

$$RT(s, s+2) \leq RT(s, s+1) + RT(s+1, s+2)$$

In order to compute a less pessimistic estimate, $RT(0, n)$, of the worst-case completion time of T^*_m, we cast the problem as one of dynamic programming, as shown in Table 7.1. In this table, the first column computes the worst-case single-mode durations, $RT(s, s+1)$. The q^{th} column computes the worst-case durations of all possible sequences of q consecutive modes, $RT(s, s+q)$, using data from the previous columns, while avoiding double-counting as we shall explain shortly. Observe that there are $n - q + 1$ rows in column q. The last column yields $RT(0, n)$, which is the solution to our problem.

1	2	.	q	.	n
RT(0,1)	RT(0,2)	.	RT(0,q)	.	RT(0,n)
RT(1,2)	RT(1,3)	.	RT(1,q+1)	.	
RT(2,3)	RT(2,4)	.	RT(2,q+2)	.	
.	.		.		
.	.		RT(n-q,n)		
.	.				
RT(n-2,n-1)	RT(n-2,n)				
RT(n-1,n)					

Table 7.1: Table illustrating the values computed using dynamic programming

Trivially, the first column can be computed from response time analysis [8], as follows:

$$RT(s, s+1) = C^*_{m,s+1} + \sum_{T^*_i \in L_{s+1}} \left\lceil \frac{RT(s, s+1)}{P_i} \right\rceil C^*_i$$

The above equation assumes that all higher-priority tasks are strictly periodic. Hence, in an interval of length t, there can be at most $\lceil t/P_i \rceil$ invocations of task T^*_i. According to reduction-based schedulability analysis [38, 42, 43], the uniprocessor tasks that result from reduction of distributed systems are strictly periodic if they arrive strictly periodically to the first shared stage with T_m (whose schedulability is being analyzed). In general, task T^*_i may have jitter J_i defined as the worst-case delay in its arrival time past the nominal beginning of its period. Thus, during a time interval, t, there may be as many as $\lceil (t + J_i)/P_i \rceil$ invocations, as shown in Figure 7.2.

The maximum response time (i.e., the content of the first column of Table 7.1) is therefore more generally

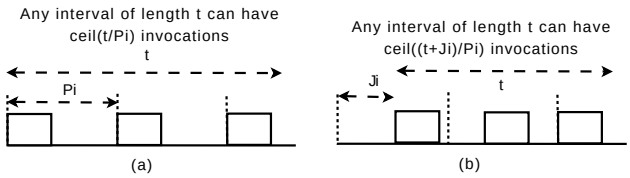

Figure 7.2: (a) System without jitter, (b) System with jitter

written as:

$$RT(s, s+1) = C^*_{m,s+1} + \sum_{T^*_i \in L_{s+1}} \left\lceil \frac{RT(s, s+1) + J_i}{P_i} \right\rceil C^*_i$$

Each subsequent column q in the dynamic programming table is computed from:

$$RT(s, s+q) = \sum_{s<j\leq s+q} C^*_{m,j} + \sum_{T^*_i} \left\lceil \frac{RT(in_i, out_i) + J_i}{P_i} \right\rceil C^*_i$$

where in_i is the larger of s and the mode after which task T^*_i enters the system, and out_i is the lower of $s+q$ and the mode after which T^*_i exits. In other words, for each task T^*_i, we compute the maximum number of invocations that can potentially delay T^*_m between (the ends of) $mode_s$ and $mode_{s+q}$. Note that, since $in_i \geq s$ and $out_i \leq s+q$, as defined above, all terms $RT(in_i, out_i)$ will have been computed from previous iterations, except the term $RT(s, s+q)$, leaving the above equation a function of $RT(s, s+q)$ on both sides, which can be solved recursively. When the dynamic programming algorithm terminates (at $q = n$), $RT(0, n)$ is returned as the final answer.

We summarize the algorithm in Figure 7.3. Let arr_i denote the mode after which T^*_i enters the system and executes together with T^*_m, and $leave_i$ denote the mode after which T^*_i leaves the system. In step 1 of the algorithm, we consider each mode independently and apply response time analysis to determine the maximum duration of each mode. In step 2, we consider q consecutive modes taken together, for increasing values of q, and determine the maximum duration of $RT(s, s+q)$ for all $s \leq n-q$, given the values of $RT(in, out)$, for all $out - in < q$. Finally, the value $RT(0, n)$ computes the worst-case response time of $T^*_{m,n}$. The correctness of the algorithm follows trivially from the fact that Equation (1) and Equation (2) never underestimate the number of invocations of higher-priority tasks that preempt T^*_m in the time intervals under consideration, and hence never underestimate the computed time intervals, including the solution $RT(0, n)$.

7.2.2 Example to Illustrate the Algorithm

We now illustrate the above algorithm using a simple example. Consider a uniprocessor system with flow-based mode changes. Let the system have 5 modes and 7 periodic tasks $T_1, T_2, \ldots, T_7$, in decreasing priority order (the corresponding distributed system is shown in Figure 7.4. We are interested in computing the

Algorithm

1 For $s = 0$ to $n-1$

1.1 Set $RT(s, s+1)^{(0)} = C^*_{m,s+1}$, $k = 0$

1.2 Repeat: Increment k

$$RT(s, s+1)^{(k)} = C^*_{m,s+1} + \sum_{T^*_i \in L_{s+1}} \left\lceil \frac{RT(s, s+1)^{(k-1)} + J_i}{P_i} \right\rceil C^*_i \tag{7.1}$$

Until $RT(s, s+1)^{(k)} = RT(s, s+1)^{(k-1)}$

1.3 Let $RT(s, s+1) = RT(s, s+1)^{(k)}$

2 For $q = 2$ to n

2.1 For $s = 0$ to $n - q$

2.2 Set $RT(s, s+q)^{(0)} = \sum_{s<j\leq s+q} C^*_{m,j}$, $k = 0$

2.3 Repeat: Increment k

$$RT(s, s+q)^{(k)} = \sum_{s<j\leq s+q} C^*_{m,j} + \sum_{T^*_i} \left\lceil \frac{RT(in_i, out_i) + J_i}{P_i} \right\rceil C^*_i, \tag{7.2}$$

where $in_i = \max(s, arr_i)$, and $out_i = \min(s+q, leave_i)$
Until $RT(s, s+q)^{(k)} = RT(s, s+q)^{(k-1)}$

2.4 Let $RT(s, s+q) = RT(s, s+q)^{(k)}$
End for

End for

3 Return $RT(0, n)$

Figure 7.3: Algorithm for analysis of a uniprocessor with flow-based mode changes

response time of the lowest priority task T_7. Tasks T_6 and T_7 operate during all modes. Tasks $T_1, T_2, \ldots, T_5$, each execute at one of the 5 modes. In particular, $mode_j$, $1 \leq j \leq 5$, has tasks T_j, T_6, and T_7. For simplicity, let us assume that all tasks are strictly periodic and have no input jitter. Let all the higher priority tasks, $T_1, T_2, \ldots, T_6$, have a unit execution time per period, and let T_7 have an execution time of 0.5 in each mode. Let the period (equal to the deadline) of T_6 be 100 units, T_7 be 200 units, and $T_1, T_2, \ldots, T_5$ be 5 units. The above task set is summarized in the table below.

Task	C_i	P_i	Mode
T_1	1	5	1
T_2	1	5	2
T_3	1	5	3
T_4	1	5	4
T_5	1	5	5
T_6	1	100	All
T_7	0.5/mode	200	All

Table 7.2: An example task set

We first compute $RT(s, s+1)$, for every s, using response time analysis. We obtain $RT(s, s+1) = 2.5$,

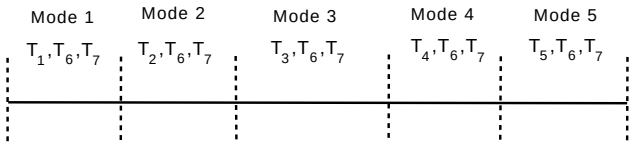

Figure 7.4: Example system

for every s, as $2.5 = 0.5 + \lceil 2.5/5 \rceil + \lceil 2.5/100 \rceil$. Next, we consider two consecutive modes taken together. For every s:

$$RT(s, s+2) = 0.5 + 0.5 + \left\lceil \frac{2.5}{5} \right\rceil + \left\lceil \frac{2.5}{5} \right\rceil + \left\lceil \frac{RT(s, s+2)}{100} \right\rceil$$

Solving, we get $RT(s, s+2) = 4$. Similarly:

$$RT(s, s+3) = 0.5 + 0.5 + 0.5 + \left\lceil \frac{2.5}{5} \right\rceil + \left\lceil \frac{2.5}{5} \right\rceil + \left\lceil \frac{2.5}{5} \right\rceil + \left\lceil \frac{RT(s, s+3)}{100} \right\rceil$$

Solving, we get $RT(s, s+3) = 5.5$. Proceeding similarly, we obtain $RT(s, s+4) = 7$, and $RT(0, 5) = 8.5$. Therefore, the end-to-end response time of T_7 is computed as 8.5. In this simple example, the bound is tight. Indeed, T_7 has a total execution time of 2.5 over the five modes, and can be preempted by at most one invocation of each higher-priority task (of 1 unit of execution time each). This adds up to 8.5 units. In general, our bound is not tight. Pessimistic estimates are possible.

7.2.3 Time Complexity of the Algorithm

The time complexity of the algorithm is certainly pseudo-polynomial. For an n stage pipeline, the number of $RT(in, out)$ terms that need to be computed is $n + (n-1) + (n-2) + \ldots + 1 = n(n-1)/2$, with each term requiring a recursive computation until convergence. This is the price we pay for the improved accuracy in determining the end-to-end response time of tasks. However, it must be noted that the iterative algorithm needs to be executed for only one equivalent hypothetical uniprocessor, unlike techniques such as [93, 23] that attempt to construct the entire schedule for all the stages in the distributed system, in order to determine the worst-case end-to-end response time.

7.3 End-to-End Delay Analysis of Distributed Tasks

In this section, we show how the end-to-end delay analysis of a distributed task can be reduced to that of analyzing a hypothetical multi-modal uniprocessor, modeled in Section 7.1. First, in Section 7.3.1, we briefly describe the distributed task model considered in this work. We present the transformation and show how

it works in Section 7.3.2.

7.3.1 Distributed System Model

The distributed system model considered in this work is similar to the model assumed in [43], and we describe it briefly here for convenience. The system running fixed priority preemptive scheduling, consists of N resource nodes $S_1, S_2, \ldots, S_N$, and a set of M periodic tasks, $T_1, T_2, \ldots, T_M$, ordered by decreasing priority. Each task requires service at some sequence of resources and must complete execution on all resources before a pre-specified end-to-end deadline. Task paths can be cyclic, that is, a task can revisit a resource multiple times before leaving the system.

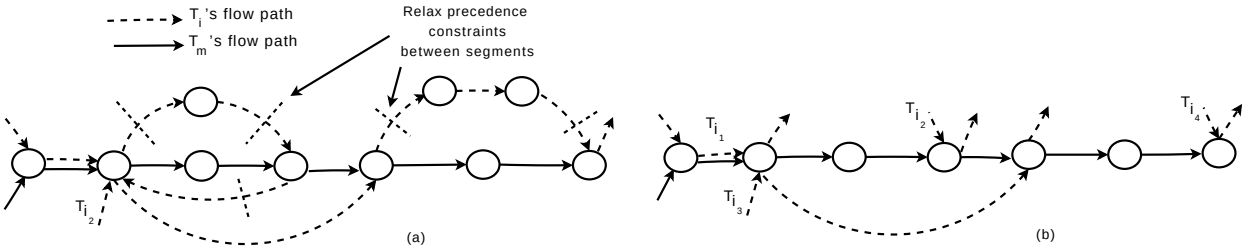

Figure 7.5: (a) Example system showing tasks T_i and T_M, (b) After relaxing constraints between different segments of T_i

Let T_M be the task whose end-to-end delay is to be estimated. Note that, a task T_i can delay T_M only along execution nodes it shares in common with T_M. As in Chapter 5, we define a task segment T_i^x as T_i's execution on a sequence of consecutive nodes along its path that is also traversed by T_M either in the same order or exactly in reverse order. We ignore the precedence constraints between different segments of each higher priority task T_i, and consider each segment as an independent task. This procedure is illustrated in Figure 7.5.

Let $C_{i,j}$ denote the worst-case execution time of an invocation of T_i on stage j, and let D_i and P_i denote the end-to-end deadline and invocation period of T_i, respectively. Let $C_{i,max}$ denote the maximum computation time of T_i across all the stages on which it executes, and let $Node_{j,max}$ denote the maximum computation time over all tasks that execute on stage j. We assume that $D_i \leq P_i$, for all i.

7.3.2 Distributed System Transformation to an Equivalent Uniprocessor with Mode Changes

As mentioned in Section 7.3.1, we consider each higher priority task segment as independent. After relaxing the precedence constraints, the system can now be thought of as having an arbitrary set of $m-1$ higher priority tasks ($m \geq M$, as we are breaking each task into multiple independent task segments), each executing

on a sequence of consecutive common stages with the lowest priority task T_m, whose worst case end-to-end delay is to be estimated. According to the delay composition theorem, each higher priority task segment T_i^x contributes a delay of at most twice its maximum stage computation time, i.e., $2C_{i,max}$ to the end-to-end delay of T_m. Apart from the per-task delay, T_m also experiences a delay component that is additive across the stages on which it executes. For each stage on which it executes, it experiences a delay that is bounded by the maximum computation time of any task on that stage, namely $Node_{j,max}$. Let Seg_i denote the set of all task segments of task T_i. The end-to-end delay of a job J_m is bounded as:

$$Delay(J_m) \leq \sum_i \sum_{J_i^x \in Seg_i} 2C_{i,max}^x + \sum_{j \in p_m} Node_{j,max} \tag{7.3}$$

Let n denote the number of stages in the path of T_m. For simplicity, we renumber the stages on which T_m executes as $S_1, S_2, \ldots, S_n$. The reduction of the distributed system to a multi-modal uniprocessor system is conducted as follows:

- Corresponding to each of the stages on which T_m executes, we create a sequence of n jobs $T_{m,1}^*, T_{m,2}^*, \ldots, T_{m,n}^*$ on the uniprocessor with the same priority as T_m, with computation times equal to $Node_{j,max}$, for $j \leq n-1$, and equal to $C_{m,max}$, for the n^{th} stage. $T_{m,j}^*$ is ready to execute on the uniprocessor right when $T_{m,j-1}^*$ completes execution, for $j \geq 2$. $T_{m,1}^*$ is ready to execute at time zero.

- Corresponding to each higher priority task T_i that executes between stages S_j and S_k, we create a uniprocessor task T_i^* with the same priority and period as T_i, and with computation time equal to $2C_{i,max}$, which is twice the maximum stage computation time of T_i. T_i^* is ready to execute when $T_{m,j-1}^*$ completes execution (or at time zero if $j = 1$) and is removed from the uniprocessor system when $T_{m,k}^*$ completes execution.

Thus, time instants where $T_{m,j}^*$, $j < n$, complete execution, act as instants of mode change in the system. A set of tasks may leave the system at this instant, and a new set of tasks may enter. Let L_j denote the set of higher priority tasks that execute together with $T_{m,j}^*$ on the uniprocessor during $mode_j$ (these are the set of tasks that execute together with T_m on stage S_j in the distributed system). From when the system starts execution at time zero, we are interested in the worst case time at which $T_{m,n}^*$ completes execution. We presented an iterative solution that was shown to converge to an upper bound on the worst-case response time of $T_{m,n}^*$, in Section 7.2. We shall now show that this indeed provides an upper bound to the worst-case end-to-end delay of T_m in the distributed system.

According to the delay composition theorem, T_m experiences a delay component of $Node_{j,max}$, the maximum computation time of any task on stage j, for each stage on which it executes. Further, in addition

to this stage-additive component, each invocation of a higher priority task segment T_i delays T_m by at most twice its maximum stage computation time, i.e., $2C_{i,max}$. Observe that, invocations of T_i can delay T_m only on stages where both T_i and T_m execute. Corresponding to the stage-additive component of the T_m's delay, each mode in the uniprocessor concludes with the execution of the lowest priority task in the mode, namely $T^*_{m,j}$, with computation time $Node_{j,max}$. Each of the busy execution intervals I_j of T_m (for stage j) consists of a set of higher priority task executions. The set of higher priority tasks that execute on stage j in the distributed system is the same as L_j on the uniprocessor. The duration of higher priority executions on I_j is upper bounded by the executions of tasks in L_j on the uniprocessor. We therefore have, $RT(0,j) \geq \sum_{k \leq j} length(I_k)$, for all $j \geq 1$. The worst-case response time of $T^*_{m,n}$ on the uniprocessor, thus provides an upper bound on the worst-case end-to-end delay of T_m.

7.3.3 An Example

In this section, we illustrate the advantage of using the analysis presented in this chapter over previous reduction based analysis techniques, using an example. By reducing the distributed system to a single static set of periodic tasks on the uniprocessor, our earlier analysis assumed that each higher priority periodic task interferes with a lower priority task throughout its execution. However, in the original distributed system, they interfere only at stages where both tasks execute together. Thus, for the case of periodic tasks, this reduction is pessimistic as it does not take into account the set of stages where a task T_i can delay T_m. Therefore, by modeling stage transitions of a task in the distributed system as mode changes in the equivalent uniprocessor, we enhance the system model with information regarding when each higher priority task T_i interferes with a lower priority task T_m under consideration.

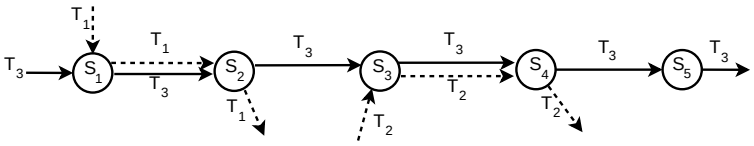

Figure 7.6: Example system

For instance, consider the example system shown in Figure 7.6. It consists of five resource stages $S_1..S_5$ and three tasks $T_1..T_3$ in decreasing priority order. T_1 executes on S_1 and S_2, T_2 executes on S_3 and S_4, and T_3 executes on all five stages. For simplicity, let us assume that the computation times of all tasks on all stages is equal to one time unit, and the end-to-end deadline of all tasks is equal to their period. Let T_1's period (same as its deadline) be 5 time units, that of T_2 be 10 time units, and that of T_3 be 12 time units. According to the reduction presented in [43], T^*_3 has a computation time of 5 units, and T^*_1 and T^*_2

have a computation time of 2 units. T_1^* has a period of 5 units, and all invocations of T_1^* that arrive before T_3^* completes execution of 5 time units, delay T_3^* in the uniprocessor.

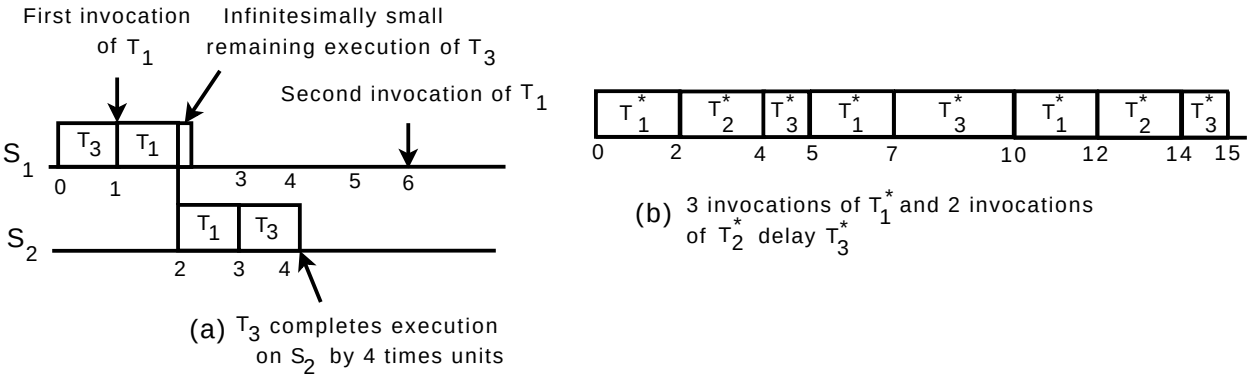

Figure 7.7: (a) Worst-case execution on S_1 and S_2 in distributed system, (b) 3 invocations of T_1^* delay T_3^* on the uniprocessor

As T_1 and T_3 are the only tasks that execute on S_1 and S_2 in the distributed system, T_3 will complete execution on stage S_2 no later than 4 time units (2 for executing T_1 and 2 for executing T_3) after T_3 arrives at stage S_1. Thus, at most one invocation of T_1 may delay T_3 in the distributed system. This is shown in Figure 7.7(a). In contrast, in the hypothetical uniprocessor, in fact, 3 invocations of T_1^* are accounted as delaying T_3^*, thus significantly overestimating the worst-case delay (see Figure 7.7(b)). Using the analysis presented in this chapter, T_1^* leaves the system after $mode_2$. The worst case response times of $mode_1$ and $mode_2$ taken independently, namely $RT(0,1)$ and $RT(1,2)$ are first calculated as 3 time units each. Next, $RT(0,2)$ is calculated as $1 + 1 + 2\lceil RT(0,2)/5 \rceil = 4$ time units. By accurately estimating the maximum duration for which T_1^* executes together with T_3^*, only one invocation of T_1^* is accounted for as delaying T_3^*.

7.4 Evaluation

In this section, we evaluate the performance of the end-to-end delay analysis technique for distributed systems proposed in this chapter. We compare with three other existing analysis techniques, namely, holistic analysis [89], network calculus [18, 19], and the meta-schedulability test presented in Chapter 5. The response time analysis technique presented in [8], is used as the uniprocessor test for the meta-schedulability test. We use a custom-built simulator with an admission controller for each of the four schedulability analysis techniques.

We consider an acyclic distributed system consisting of N resource stages. As we are interested in the performance of large systems, the default value of N is assumed to be 20. We assume periodic tasks scheduled using a deadline monotonic scheduling policy. Simulation parameters are chosen similar to previous chapters.

The default value of node probability parameter, *node_prob*, is 0.8. The default value of the deadline ratio parameter, DR, is 2.0. We used a value of $T = 1/20$ for the task resolution parameter. In this evaluation, we assume that the initial jitter for all tasks is taken to be zero.

Each point presented in the figures below are average values obtained from 100 executions, with each execution running for 80000 task invocations. In order to ensure that the comparison is fair, the admission controllers for each of the four schedulability analysis techniques are allowed to run on the same 100 task sets. The 95% confidence interval of the values presented are within 1% of the mean, and are not shown in the figures for the sake of legibility.

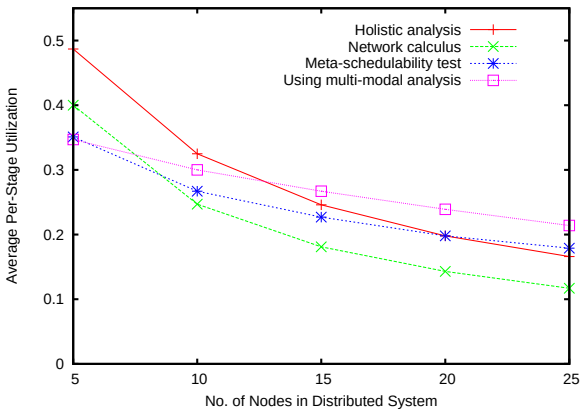

Figure 7.8: Comparison of average per stage utilization for different number of stages in the system

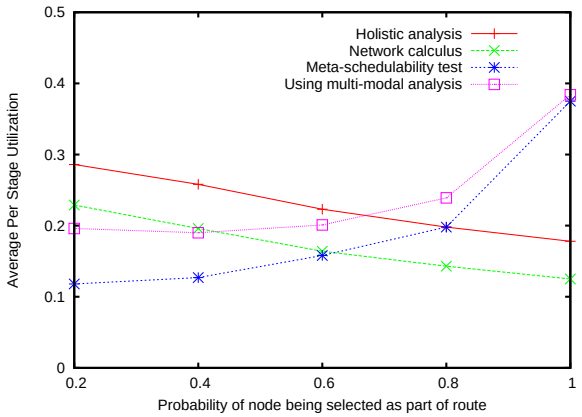

Figure 7.9: Comparison of average per stage utilization for different probabilities of node being part of a task's route

First we compare the four schedulability tests for the admissible utilization for different number of nodes in the system, the results of which are shown in Figure 7.8. We consider system sizes ranging from 5 nodes to 25 nodes. Both network calculus and holistic analysis perform poorly when the system size increases, and the drop in their admissible utilization is steeper than for the meta-schedulability test and the multi-modal analysis presented in this chapter. We note that for small system sizes (up to 10 nodes), holistic analysis in fact, performs better than the multi-modal analysis. The reduction from the distributed system to the multi-modal uniprocessor assumes that each higher priority task invocation delays the lowest priority task at two stages (according to the delay composition theorem). However, not all higher priority task invocations interfere at two stages, and some cause a delay less than what is quantified by the delay composition theorem as the worst-case. For small system sizes, holistic analysis is able to determine the number of invocations of higher priority tasks that delay the lowest priority task under consideration more accurately. However, for large systems (more than 15 nodes), holistic analysis becomes extremely pessimistic and the multi-modal analysis performs better. The multi-modal analysis is able to admit about 25% more tasks than the next best analysis technique for large systems.

We next varied the probability with which a node is chosen to be part of a task's route (the value *node_prob*), and present the results in Figure 7.9. As the value of *node_prob* increases, task routes become longer, and both holistic analysis and network calculus become more pessimistic and their admissible utilization drops. The meta-schedulability test and its extension presented in this chapter perform well for tasks with long routes, as the number of precedence constraints between successive stages of tasks that are relaxed become lower. For tasks with short routes, a larger fraction of the total number of constraints are relaxed leading to poorer performance. Thus, for both these tests, it is the fraction of precedence constraints that are relaxed that affects performance and not the length of task routes, unlike holistic analysis and network calculus. Further, when the *node_prob* value is small and tasks have short routes through the system, the problem with the meta-schedulability test explained in Section 7.3.3 becomes exacerbated. Higher priority periodic tasks delay a lower priority task only at a few stages. However, in the hypothetical uniprocessor system, the corresponding tasks are assumed to delay the lower priority task throughout its execution. This problem is overcome by the multi-modal uniprocessor model presented in this chapter. In fact, for short task routes, the extension allows almost twice as many tasks to be admitted as compared to the meta-schedulability test. For strict pipelines (a *node_prob* value of 1), the analysis in this chapter admits more than twice as many tasks as holistic analysis or network calculus admits. The test accurately estimates the parallelism in the execution of successive stages in the pipelined distributed system, and is able to perform significantly better than holistic analysis.

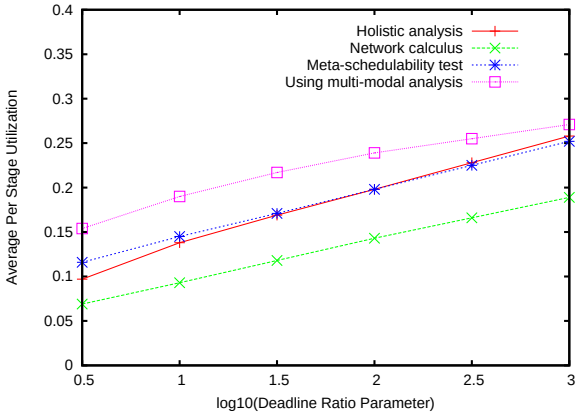

Figure 7.10: Comparison of average per stage utilization for different deadline ratio parameter values

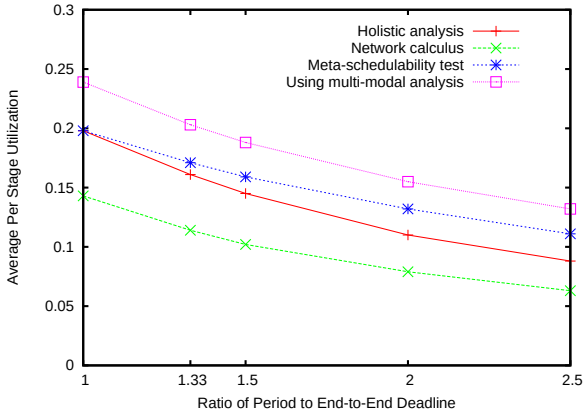

Figure 7.11: Comparison of average per stage utilization for different ratios of task periods to end-to-end deadlines

Next, we compare the schedulability tests for different deadline ratio parameter values. For small values of DR, the computation times of all the tasks are comparable. For larger values of DR (closer to 3), the computation times of tasks are widely varying and lower priority tasks manage to execute in the background of higher priority tasks. This allows busy periods to be longer and the utilization of the system to be higher

for all the schedulability tests. The analysis presented in this work achieves an increase of 20-50%, compared to the admissible utilization using holistic analysis.

All the experiments conducted so far assumed that the task periods are equal to their end-to-end deadline. We allowed the end-to-end deadlines to be progressively tighter and considered ratios of task periods to end-to-end deadlines to be $1.33, 1.5, 2.0$, and 2.5, while keeping the task periods the same. As expected the admissible utilization of all the four analysis techniques dropped with increasing ratio values (decreasing end-to-end deadlines). Yet, the multi-modal analysis significantly outperforms existing analysis techniques for all values of the ratio of task periods to end-to-end deadlines.

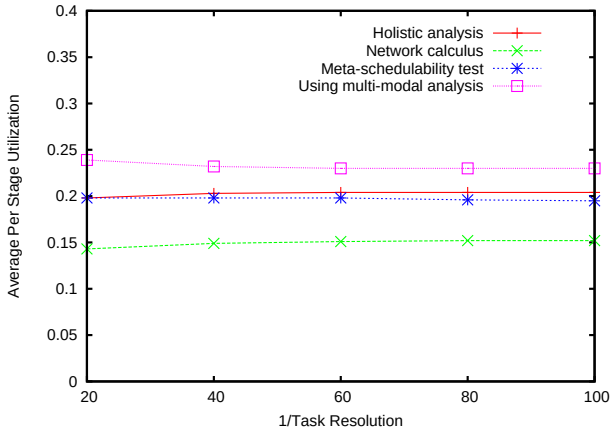

Figure 7.12: Comparison of average per stage utilization for different task resolution parameter values

Finally, we conducted experiments that varied the task resolution parameter values, that is, the ratio of the computation times of tasks to their end-to-end deadline. The average per stage utilization for the four admission control tests for task resolution parameter values of $1/20, 1/40, 1/60, 1/80$, and $1/100$ are shown in Figure 7.12 (note that the x-axis shows 1/task resolution). The task resolution parameter does not affect the performance of the various tests, showing that the tests are not sensitive to the size of the tasks. This is due to the preemptive nature of scheduling. A task resolution parameter of $1/100$ would approximately have five times as many tasks admitted as a task resolution parameter of $1/20$, but each task would be five times smaller in terms of computation time, overall resulting in approximately the same interference to the lowest priority task.

Chapter 8

Structural Robustness of Distributed Real-Time Systems Towards Uncertainties in Service Times

With applications becoming more complex and requiring larger system capacity to function, the emphasis is shifting towards increasing distribution. Real-time applications are growing in scale, both in terms of the number of tasks involved as well as the number of resources. Such large and complex distributed systems typically execute soft real-time applications, where there is significant uncertainty in the execution times of tasks on individual resources, or the worst-case timing is not entirely verified. An extremely important problem in such systems with uncertainties in the worst-case execution times of tasks, is how do we optimize the allocation of resources to individual execution stages of tasks (the topology of the system) to minimize the effect that the uncertainties have on the end-to-end delay of tasks. The problem applies to systems where tasks are described as flow paths with end-to-end time constraints. Each flow path is made of sub-tasks called stages with presumed per-stage worst-case computation times, that can potentially be violated.

Towards addressing this problem, in this chapter, we define a metric called *structural robustness* that measures the robustness of the end-to-end timing behavior of a system's task flow graph towards unexpected violations in the worst-case application execution times on individual resources. We demonstrate that by efficiently allocating resources to execution stages of end-to-end tasks, the flow paths of tasks can be optimized to improve the system's structural robustness. Given a particular system configuration, the structural robustness metric computes a notion of end-to-end weighted task lateness with respect to individual worst-case execution times of tasks on stages. We also present a simple hill climbing algorithm that can be used to explore the space of all system configurations to determine a highly robust configuration. We show using simulations that this algorithm is able to reduce the number of deadline misses by over 50% by finding robust system configurations in the presence of unexpected execution time violations.

Our algorithm to improve the structural robustness of systems builds on our delay composition results derived in earlier chapters, which show that not all executions of tasks on individual stages affect the worst-case end-to-end delays of tasks. By altering the topology of the system, we reduce the number of stage executions of tasks that affect the end-to-end delay of other tasks in the system, thereby reducing the sensitivity of the end-to-end delays of tasks to individual task executions on stages.

There may be several reasons for unanticipated variations in application execution times on individual stages. First, in applications such as automobile systems, it is extremely difficult to accurately determine the worst-case execution times of tasks, especially early in the design life cycle, and there could be errors in estimation. Second, in settings such as wireless networks, variations in link quality and interference effects could significantly impact the worst-case packet transfer times (execution times). Third, in many systems, the worst-case execution times could be significantly higher than the most common case and may occur extremely rarely (possibly due to faults within the system). For such systems, in order to improve performance it might be more prudent to consider lower, more common estimates of the execution times for end-to-end delay calculations, and have mechanisms to cope with uncertainties in the execution times.

The rest of the chapter is organized as follows. In Section 8.1, we formally define the structural robustness metric. We describe the system model in Section 8.2. We provide an overview and intuition for how we optimize the system topology using the structural robustness metric in Section 8.3. In Section 8.4, we describe how the structural robustness metric is calculated, and how it can be easily recomputed for a single change in the system configuration. We also describe the hill climbing algorithm to improve the robustness of a given system towards unanticipated violations in the worst-case stage execution times of tasks. We evaluate our proposed technique using simulations in Section 8.5.

8.1 Structural Robustness

The robustness of a system is mainly affected by the degree of interactive complexity between tasks in the system. In order to measure the structural robustness of a system, we seek to quantify the degree of interactive complexity between tasks. As we are interested in ensuring that end-to-end timing constraints of tasks are not violated, we specifically study the complexity of temporal interactions within the system. In other words, we are interested in estimating the extent to which task execution times on individual stages affect the worst-case end-to-end delays of tasks.

We consider systems with tasks described as flow paths (a path may consist of just one resource). Tasks require execution at a sequence of resources along its path (each called a stage execution) and the end-to-end execution must complete within certain pre-specified time constraints. Resources may be complex and represent entire subsystems. Each stage execution of a task consists of the task's execution on one such resource. We assume that the worst-case extent to which the execution of task i on resource j affects the end-to-end delay of any task in the system is known as $X_{i,j}$. The worst-case delay $X_{i,j}$ is a function of the worst-case execution time $C_{i,j}$ of task i on resource j, and will depend on how the resource is scheduled. For instance, when the resource consists of only one resource scheduled in priority order, $X_{i,j}$ may be equal to

$C_{i,j}$. If the resource serves tasks based on a TDMA schedule, then $X_{i,j}$ may be larger than $C_{i,j}$, as it may take several TDMA cycles before the execution of task i completes on the resource j. If parallelism exists in the execution of tasks within the resource, $X_{i,j}$ may be lesser than $C_{i,j}$. Further, it is possible for these worst-case delay estimates to be violated. Let the number of resources in the system be N, and the number of tasks be M. Let Q_k denote the set of tuples (i, j), such that an infinitesimal increase in the worst-case execution time of task i on resource j would result in the worst-case end-to-end delay of task k to increase (task i can be the same as task k).

In order to determine a single structural robustness metric for the entire system, we first estimate the extent of temporal interactions within the system. This is estimated by computing the effect that a particular stage execution of a task i has on the worst-case end-to-end delay of a task k, $X_{i,j}$, weighted by the importance of task k, and accumulated across all tasks i and k. Although two tasks i and k both execute at a resource, it is possible that they do not affect each other's worst-case end-to-end delays. As we are interested in the extent of temporal interactions within the system, we normalize the above computed value with respect to the product of the total of all $X_{i,j}$'s and the total of all $I(k)$'s of tasks. As a larger value for the extent of temporal interactions within the system reflects a lower level of robustness, we compute the structural robustness metric by considering one minus the above normalized value. We formally define *structural robustness* as follows:

Definition: Given an importance vector I that denotes the relative importance of a task with respect to other tasks in the system, the *structural robustness* of a particular system's task flow graph is defined as:

$$\omega = 1 - \frac{\sum_{k \leq M} \sum_{(i,j) \in Q_k} X_{i,j} I(k)}{\sum_{k \leq M} I(k) \sum_{(i,j)} X_{i,j}} \tag{8.1}$$

Let us take a closer look at the structural robustness metric defined in Equation 8.1. Note that, the definition of structural robustness is particularly concerned with task executions on individual stages that contribute towards the worst-case end-to-end delays of other tasks. Individual stage executions of tasks that affect the worst-case end-to-end delays of a larger number of tasks or those that affect more important tasks are weighted more. For instance, a stage execution of a task A that affects the worst-case end-to-end delay of one other task, contributes less towards reducing the structural robustness of the system than a stage execution of a task B that affects several other tasks. Further, a stage execution of a task A that affects the worst-case end-to-end delay of another task by X, contributes less towards reducing the structural robustness of the system than a stage execution of a task B that affects another task by, say $10X$ (the temporal interaction is more due to task B than due to task A in both instances). This explains why the metric accumulates the effect that stage executions have on the end-to-end delays of other tasks.

The importance vector is specified by the application and reflects whether missing a deadline for one task is more tolerable than missing a deadline for another task. For instance, for tasks that are homogeneous, a simple importance vector could be to assign an equal importance to all the tasks (a value of 1 for each entry of the vector). Alternatively, the importance of each task could be assigned to be inversely proportional to its deadline. In essence, the value associated to each task reflects its importance towards the application's correctness and performance.

The problem we address in this chapter is to assign tasks to resources so as to maximize the above defined structural robustness metric. Such an optimized system would be less sensitive to unanticipated delays in particular stage executions of tasks and would minimize the number of deadline misses, as it reduces the extent of temporal interactions within the system. In Section 8.2, we define the particular system model we consider in this chapter. We envision that future work will enhance the scope of systems that are optimized for structural robustness to unanticipated delays in stage execution times.

8.2 System Model

We consider a distributed system comprising of N different kinds of resources, $R_1, R_2, \ldots, R_N$. Each resource R_i has $r_i \geq 1$ identical instances of the resource available within the system. A resource can be anything that serves tasks in a fixed priority preemptive scheduling order (e.g., processor, communication link). Let N_{tot} denote the total number of all instances of resources present in the system, and let the instances be arbitrarily named $S_1, S_2, \ldots, S_{N_{tot}}$. The system serves M end-to-end soft real-time tasks, $T_1, T_2, \ldots, T_M$, ordered by decreasing priority. Each task T_i requires execution on a pre-specified sequence of resources and must complete execution on all resources before a pre-specified end-to-end deadline. The relative priority of each task remains the same across all the resources on which it executes. When multiple instances of a resource are available, any one of the instances can be assigned to serve a task requesting that resource. Each resource instance at which a task executes is referred to as a stage. For ease of exposition, we assume that the union of all task paths forms a Directed Acyclic Graph (DAG). Later, in Section 8.4.2, we show how our technique can be easily extended to handle cycles in the task paths (tasks can revisit resources multiple times). Tasks may be periodic or aperiodic.

Let $C_{i,j}$ denote the estimated worst-case execution time of an invocation of T_i on a resource instance j, and is the same regardless of which instance of the resource is assigned to serve it. Each resource group has only one resource, and hence the extent of the delay that the execution of a task T_i on resource j can cause another task, $X_{i,j}$, is the same as $C_{i,j}$. Although we have an estimate of the worst-case execution time, we consider it possible for tasks to exceed this estimate. Let D_k denote the end-to-end deadline of T_k.

Given such a system, the objective is to assign tasks to resource instances as per their resource requirements, so as to minimize the number of deadline misses within the system in the presence of unanticipated delays in execution times. The algorithm presented in this work achieves this objective by reducing the sensitivity of the end-to-end timing behavior of the system's task flow graph to specific execution times and not allowing any spikes in the execution times to propagate to the worst-case end-to-end delay.

A particular assignment of tasks to instances of resources requested by it, is termed as a configuration. The sequence of stages followed by a task T_i in a configuration C, is denoted by $Path_i^C$. Let $C_{i,max}$ denote the maximum computation time of T_i across all the stages on which it executes, and let $Node_{j,max}$ denote the maximum computation time over all tasks that execute on a resource instance j.

Note that, a task T_i can delay T_k only along execution stages it shares in common with T_k. We define a task segment T_i^x (the segments are indexed) as T_i's execution on a sequence of consecutive resource instances along its path that is also traversed by T_k either in the same order or exactly in reverse order. Let $C_{i,max}^x$ be the maximum computation time of T_i across all stages in segment T_i^x, and let resource instance j be the stage corresponding to the maximum computation time, also referred to as the max-stage of the segment. We ignore the precedence constraints between different segments of each higher priority task T_i, and consider each segment as an independent task. As explained in Chapter 4, note that this does not decrease the end-to-end delay of T_k as we only remove certain precedence constraints, thereby increasing the set of possible arrival patterns of tasks to stages. Thus, our delay bound estimate errs on the safe side.

8.3 Solution Overview

In this section, we present the main idea and intuition behind optimizing the task paths to improve the structural robustness of the system towards unanticipated delays in the execution times of tasks. By changing how tasks are assigned to resource instances, we reduce the sensitivity of the worst-case end-to-end delays of tasks to the individual execution times. That is, we ensure that each higher priority task execution on a stage affects the worst-case end-to-end delay of fewer lower priority tasks.

We first need to reflect on our earlier work on quantifying the worst-case end-to-end delay of a job in terms of the computation times of higher priority jobs that execute together with it. As it is extremely difficult to accurately quantify the actual delay of tasks, we work with worst-case end-to-end delay bounds of tasks for the purposes of studying the structural robustness of systems. The delay composition theorem provides such an end-to-end delay bound.

Recall that, according to the theorem, for each higher priority task segment, only the maximum stage execution time over all stages belonging to the segment contributes to the worst-case end-to-end delay bound.

This maximum stage execution time is referred to as a *delay term* and the corresponding stage is referred to as a *max-stage*. Further, for each stage j, a maximum computation time across all higher or equal priority jobs executing on that stage, $Node_{j,max}$, figures in the delay expression. This term is referred to as the stage-additive component as it is additive across the stages on which J_1 executes and is independent of the number of jobs in the system. Thus, unanticipated delays for jobs in such non-maximum stages do not affect the worst-case end-to-end delay of lower priority jobs (as long as they do not exceed the maximum stage computation time).

Thus, due to the distributed nature of computation and the overlap in the execution of different stages, the system naturally has a certain tolerance towards unanticipated delays as long as these delays do not occur at the stage executions that feature in the delay terms of the delay composition theorem. The objective of the robustness optimization we perform, is to reduce the number of such terms in the end-to-end delay bounds of jobs. This can be done in multiple ways. First, moving a task from one resource instance to another, could eliminate interference due to a higher priority job segment to a lower priority job. Figure 8.1(a) illustrates this scenario. Either the higher priority or the lower priority task can be moved away to avoid the interference.

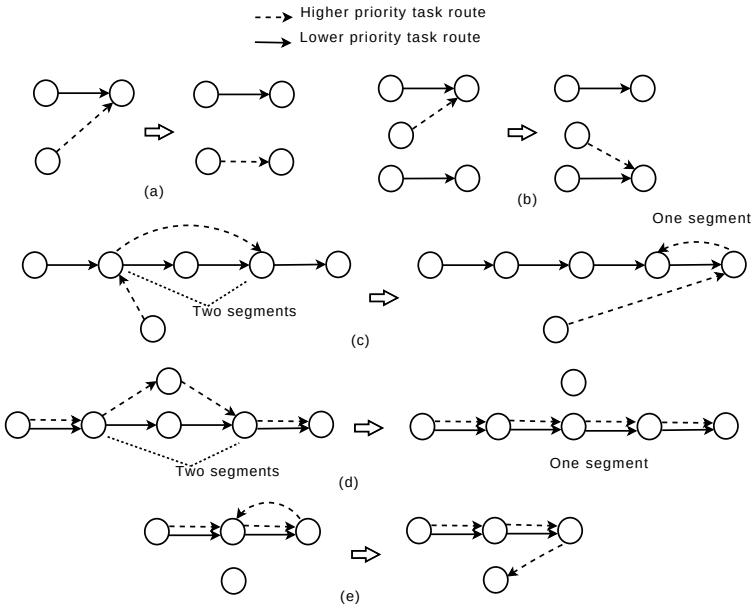

Figure 8.1: Example system showing two tasks and how various transformations can reduce the number of terms in the worst-case end-to-end delay bounds

It is however very likely, as shown in Figure 8.1(b), that when a task is moved from an instance j to another instance j' of the same resource, a different set of tasks are scheduled to execute on j'. Thus, moving a task from j to j' can cause some interferences to be removed at stage j and other new ones involving a different set of tasks to be created at stage j'. The importance vector as defined in Section 8.1,

enables us to estimate and compare the structural robustness of the system under different configurations. In Section 8.4, we discuss in further detail how the structural robustness of different system configurations can be quantitatively estimated using the importance vector.

Moving a task from one resource instance to another could reduce the number of delay terms in other ways. As shown in Figures 8.1(c) and 8.1(d), it could combine multiple segments into a single segment. When segments are combined into one, only one delay term for the entire combined segment needs to be accounted for in the delay bound, as against a delay term for each of the original segments.

Moving tasks between different instances of resources can help load balancing the system. Moving a task from one instance to a less utilized instance, could reduce the delay for a large number tasks at the expense of increasing the delay for a few tasks which are well within their deadline stipulations. Load balancing the system will improve the system's robustness, as fewer tasks will be affected by unanticipated delays in the executions at a particular resource instance.

Finally, as illustrated in Figure 8.1(e) for tasks that may contain cycles in their path, the number of segments can be reduced by relaxing the loops. In the example shown, as the higher priority task revisits a node, two segments of the higher priority task delay the lower priority task. Once the loop is relaxed, only one segment of the higher priority task delays the lower priority task.

Thus, by intelligently moving tasks around between instances of resources we can reduce the sensitivity of the worst-case end-to-end delays of tasks to the individual stage execution times. When there are unanticipated delays at certain executions, they are then much less likely to propagate to affect the worst-case end-to-end delays.

8.4 Methodology to Improve Structural Robustness of the System

In this section, we present our methodology and algorithm to improve the structural robustness of distributed systems to unanticipated delays in the stage execution times. In Section 8.4.1, we describe the general algorithm targeted towards execution graphs that are directed and acyclic. In Section 8.4.2, we show how the algorithm can be easily extended to task paths that contain cycles.

8.4.1 General Algorithm

We first describe how we compute the structural robustness metric and how we can easily recompute the metric when the configuration is changed by moving a task executing on one resource instance onto another

instance belonging to the same resource type. Then, we describe a simple hill climbing algorithm to explore the search space of system configurations to find a highly robust configuration.

For each feasible configuration C, we define a 3-dimensional matrix $[W^C_{i,j,k}]_{M \times N_{tot} \times M}$, to store the terms in the end-to-end delay bound as per the delay composition theorem for each job J_k. We call this the *delay matrix*. All entries in the matrix are initially assigned to zero. For each job J_k and each higher priority job segment of a job J_i, there is a delay term equal to twice J_i's maximum stage computation time over all stages in that segment. Suppose, the maximum occurs at a stage j. Then, $W^C_{i,j,k}$ is assigned to $2C_{i,j}$. Further, for each stage j on which J_k executes, there is a delay term equal to one maximum stage computation time of any higher priority job that executes on it. If this maximum corresponds to a task i, then $W^C_{i,j,k}$ is incremented by $C_{i,j}$. Note that, if jobs J_i and J_k don't both execute on a stage j, or if J_i has a lower priority than J_k, then the entry $W^C_{i,j,k}$ will remain zero regardless of the system configuration.

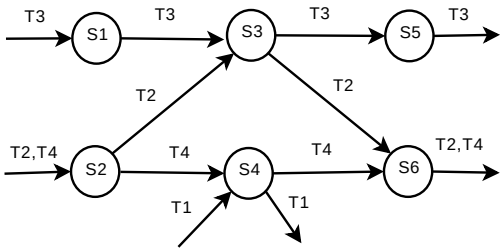

Figure 8.2: Example system with four tasks, three resource types, and two instances of each resource

Through the rest of this section, we shall use a running example of a system with three resource types $R1, R2$, and $R3$. There are two instances of each resource available: $S1$ and $S2$ of type $R1$, $S3$ and $S4$ of type $R2$, $S5$ and $S6$ of type $R3$. The system comprises of four tasks $T1$, $T2$, $T3$, and $T4$, in decreasing priority order, with their task paths as shown in Figure 8.2. Task $T1$ executes only on instance $S4$, $T2$ executes along the path $S2-S3-S6$, $T3$ has the path $S1-S3-S5$, and $T4$ has the path $S2-S4-S6$. For simplicity, let us assume that each task requires one unit execution time at each stage on which it executes. Let us denote this system configuration as C.

Let us now calculate the delay matrix for this system configuration. Task $T1$ executes only on instance $S4$, and hence delays only task $T4$. Two segments of task $T2$ delay task $T4$, at resource instances $S2$ and $S6$, respectively. Task $T2$ also delays task $T3$ at resource instance $S3$. Tasks $T3$ and $T4$ execute on mutually disjoint stages and hence do not interfere with each other. Each of the above delay terms have a value of twice the maximum stage computation time of the higher priority task segment, which is 2 time units. Further, each task experiences a delay of one maximum stage computation time across all tasks for each stage on which it executes, which equals one unit. As all the computation times of tasks are equal, this delay is accounted for as the delay to the task due to itself. The matrices for each resource instance is constructed

as follows (i^{th} row, k^{th} column, denotes the delay term that task i causes task k):

$$W^C_{S1} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, W^C_{S2} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, W^C_{S3} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix},$$

$$W^C_{S4} = \begin{bmatrix} 1 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, W^C_{S5} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, W^C_{S6} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Note that the particular entries in the matrix depend on which stages contribute to the maximum computation times for each segment, which in turn depends on the system configuration and the higher priority job segments for each job. By combining segments together, by creating segments that affect fewer lower priority jobs, or by removing loops in the task graph (as explained in Section 8.3), it is possible to reduce the number of terms in the delay bounds of all the tasks.

Note that, the structural robustness metric can be calculated based on the delay matrix as follows:

$$\omega^C = 1 - \frac{\sum_{i,j,k} W^C_{i,j,k} \times I(k)}{\sum_{k \leq M} I(k) \sum_{(i,j)} C_{i,j}} \tag{8.2}$$

A configuration that has a higher value for this metric ω is deemed to be more robust to unanticipated delays as the dependence of the worst-case end-to-end delays of tasks on individual stage computation times is lower. Let us now compute the structural robustness metric for our example system configuration. For simplicity, let us assume that the importance vector has a value of one for each task (all tasks have the same relative importance). The numerator of the fractional part of the structural robustness metric is simply the sum of all the entries in the delay matrix, which comes to 18 units. The sum of importance vectors is 4 and the sum of all $X_{i,j}$s is 10, so the denominator of the fractional part is computed as $4 \times 10 = 40$ units. Hence, the structural robustness of the system configuration is $1 - \frac{18}{40} = 0.55$.

When a task i is moved from instance j to another instance j' of the same resource leading to a new configuration C', multiple changes occur in the delay matrix. Task i no longer interferes with any lower priority tasks at instance j, and itself does not experience any interference from higher priority tasks. Therefore, all entries corresponding to the row and column of task i at instance j are set to zero. For each lower priority task k that executes at instance j, we need to check how the segment of task i that included instance j in configuration C (say, Seg^x_i) has changed due to the move. The possible cases of how Seg^x_i changes are illustrated in Figure 8.3. First, if instance j was the only stage belonging to Seg^x_i, then this

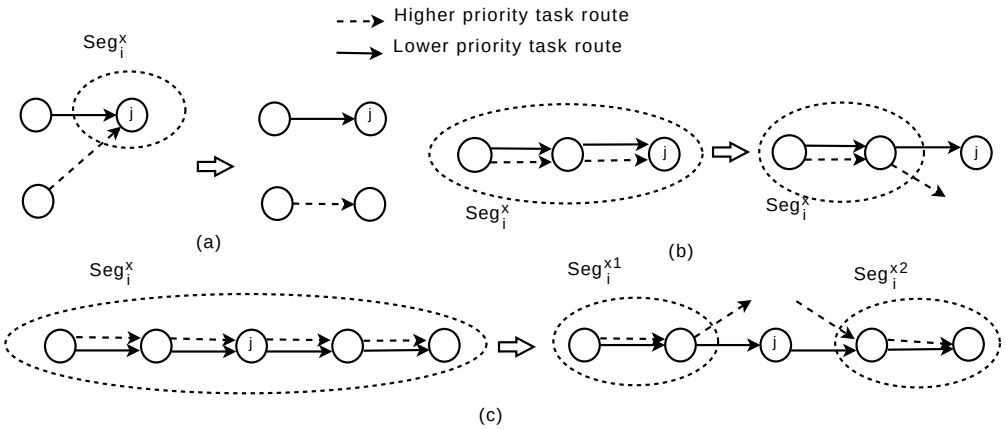

Figure 8.3: Figure illustrating the possible cases when a higher priority job is moved out of a resource instance j

segment no longer exists, as shown in Figure 8.3(a). Second, if instance j was either the first or the last stage of the segment, then the segment remains with the removal of instance j, as shown in Figure 8.3(b). If instance j was the max-stage of the original segment, we need to determine the max-stage of the modified segment. Suppose the max-stage is instance l, we need to set $W^{C'}_{i,l,k}$ to $2C_{i,l}$. If instance j was not the max-stage in the original segment, then no changes need to be made. Third, if instance j was neither the first nor the last stage of the segment (was an intermediate stage), then the move causes Seg_i^x to be split into two segments, as shown in Figure 8.3(c). We need to determine the max-stages for both the new segments of task i, and update the delay matrix entries for them, if either of them wasn't the max-stage of the original segment Seg_i^x. The above procedure for updating the delay matrix when a higher priority job i is moved out of an instance j is presented as procedure $RemoveTaskFromSegment(W^{C'}, i, j, k)$.

Similarly, for each higher priority task that executes at instance j, its interference to task i at instance j has been removed (set to zero). We need to check each higher priority task segment that originally included instance j, to see if the segment is removed, reduced by one stage, or split into two segments. The actions that need to be taken for each case are similar to the description above, and is executed by invoking $RemoveTaskFromSegment(W^{C'}, k, j, i)$.

As task i is moved to instance j', additional interferences need to be accounted for at instance j'. For each lower priority task k that executes at j', we need to consider the segment of task i that includes j' (say, Seg_i^x) in the new configuration. The possible cases of how Seg_i^x changes due to the move are illustrated in Figure 8.4. First, if Seg_i^x consists of just the instance j', then we need to set the value of $W^{C'}_{i,j',k}$ to $2C_{i,j'}$ (this is a new segment, as shown in Figure 8.4(a)). Second, if instance j' now becomes the first or last stage of an already existing segment, as shown in Figure 8.4(b), we need to check if task i's computation time on j' is larger that on the current max-stage l of the segment. If so, we need to set $W^{C'}_{i,j',k}$ to $2C_{i,j'}$ and

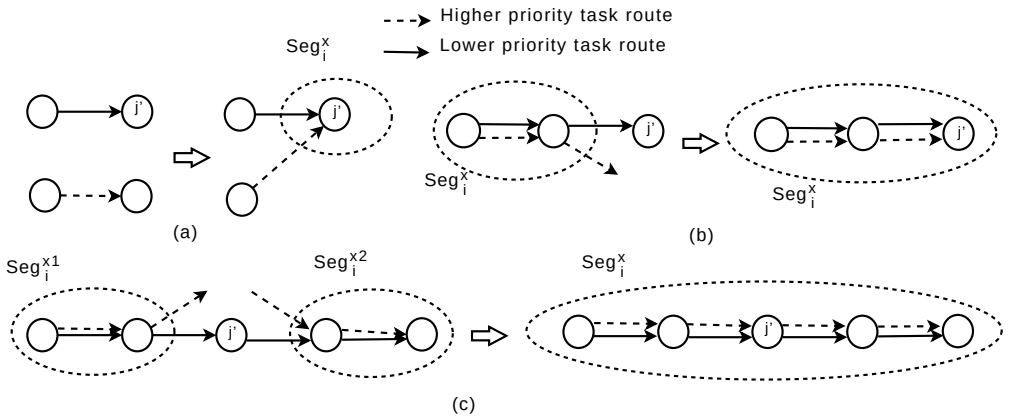

Figure 8.4: Figure illustrating the possible cases when a higher priority job is moved in to execute at instance j'

decrement $W^{C'}_{i,l,k}$ by $2C_{i,l}$. Third, task i executing on instance j' could combine two existing segments of task i as shown in Figure 8.4(c). In this case, we need to determine the new maximum stage computation time of task i on the combined segment. If the max-stage is j', we need to increment the delay matrix entry $W^{C'}_{i,j',k}$ by $2C_{i,j'}$. We then need to decrement the delay matrix entry for the stage or stages that are no longer the max-stage of the segment. The above procedure for updating the delay matrix when a higher priority job i is moved in to execute at an instance j is presented as procedure $AddTaskToSegment(W^{C'}, i, j', k)$.

Similarly, for each higher priority task k that executes at j' its interference to task i needs to be accounted for. We need to check each higher priority task segment, to see if a new segment is added, an existing segment is augmented by one stage (instance j'), or if two segments have been combined together. The actions that need to be performed for each case are similar to the corresponding cases described above, and is executed by invoking $AddTaskToSegment(W^{C'}, k, j', i)$. Finally, the stage additive component, which is the maximum computation time across all jobs with higher or equal priority to task i at instance j' (say, due to task k) needs to be updated by incrementing $W^{C'}_{k,j',i}$ by $C_{k,j'}$.

The algorithm to determine the changes in the system topology and the delay matrix when the configuration is altered by moving a task i from one instance j to another instance j' is described by the algorithm $UpdateTopology(W^C, i, j, j')$.

RemoveTaskFromSegment(W^C, i, j, k)

Comment: Updates $W^{C'}$ with respect to lower priority task k, when task i is moved out of instance j

1. If task i's segment consists only of instance j, then continue
2. If j was either the first or last stage of task i's segment and was the max-stage

   then find the new max-stage, instance l

Increment $W^{C'}_{i,l,k}$ by $2C_{i,l}$

3. If j was an intermediate stage of task i's segment

then find max-stages, l and l', of the two new segments

let l_{old} be the max-stage of the original segment

If $l_{old} \neq j$, then decrement $W^{C'}_{i,l_{old},k}$ by $2C_{i,l_{old}}$

Increment $W^{C'}_{i,l,k}$ by $2C_{i,l}$ and $W^{C'}_{i,l',k}$ by $2C_{i,l'}$

AddTaskToSegment(W^C, i, j', k)

Comment: Updates $W^{C'}$ with respect to lower priority task k,

when task i is moved in to execute at instance j'

1. If task i's segment consists only of instance j', then set $W^{C'}_{i,j',k} = 2C_{i,j'}$

2. If j' is either the first or last stage of task i's segment

If l was the previous max-stage and $C_{i,j'} > C_{i,l}$

then set $W^{C'}_{i,j',k} = 2C_{i,j'}$, decrement $W^{C'}_{i,l,k}$ by $2C_{i,l}$

3. If j' combines two segments of task i

then find max-stages, l and l', of original segments; find max-stage, l_{new}, of combined segment

increment $W^{C'}_{i,l_{new},k}$ by $2C_{i,l_{new}}$; decrement $W^{C'}_{i,l,k}$ by $2C_{i,l}$ and $W^{C'}_{i,l',k}$ by $2C_{i,l'}$

UpdateTopology(W^C, i, j, j')

Output: $W^{C'}$, for configuration C', where task i executes on instance j' instead of j

Initialize $W^{C'} = W^C$

Stage j:

1. $W^{C'}_{i,j,k} = W^{C'}_{k,j,i} = 0$, for every k.

2. For each task k of lower priority than task i: $RemoveTaskFromSegment(W^C, i, j, k)$

3. For each task k of higher priority than task i: $RemoveTaskFromSegment(W^C, k, j, i)$

Stage j':

1. For each task k of lower priority than task i: $AddTaskToSegment(W^C, i, j', k)$

2. For each task k of higher priority than task i: $AddTaskToSegment(W^C, k, j', i)$

3. Find task x such that $C_{x,j'} \geq C_{k,j'}$ for all higher or equal priority tasks k executing at j'

Increment $W^{C'}_{x,j',i}$ by $C_{x,j'}$

The complexity of the above algorithm is $O(MN_{tot})$, which is the product of the number of tasks and the number of resource instances in the system. At each of the instances j and j' we need to consider all the other tasks executing on that instance, and need to update the segments that are altered by the move. As each segment is at most as long as the number of instances in the system, the complexity of the algorithm is bounded as $O(MN_{tot})$.

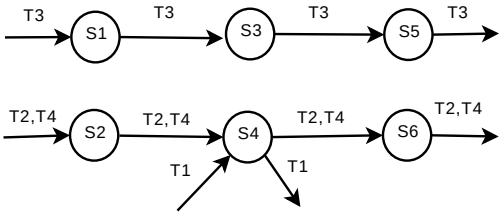

Figure 8.5: Configuration C' after moving task $T2$ from $S3$ to $S4$

Let us go back to our example system and see how we can improve the structural robustness metric by moving a task from one instance to another. First, let us move task $T2$ from $S3$ to $S4$, and let the new configuration be C', as shown in Figure 8.5. We need to first update the delay matrix to reflect the move. Task $T2$ no longer delays task $T3$ at $S3$. All terms in the row and column corresponding to task $T2$ at $S3$ are set to zero. Since, $S3$ was the only stage in the segment of task $T2$ that interfered with task $T3$, the segment is now removed. At $S4$, task $T1$ has a higher priority than task $T2$ and therefore interferes with it. As $S4$ is the only stage on which $T1$ executes, the segment has only one stage. We therefore set $W^{C'}_{T1,S4,T2}$ to 2 units. Task $T4$ has a lower priority than task $T2$. Task $T2$ executing on $S4$ causes two segments of $T2$ to be combined into a single segment. As all the computation times of $T2$ are equal to one, let us choose the maximum to be $S2$. As $S6$ is no longer the max-stage of a segment of $T2$, the term $W^{C'}_{T2,S6,T4}$ is set to zero. Finally, the stage-additive component for $T2$ needs to be set at $S4$ and $W^{C'}_{T2,S4,T2}$ is set to 1 unit. The updated matrices for each resource instance for the new configuration C' are as follows:

$$W^{C'}_{S1} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, W^{C'}_{S2} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, W^{C'}_{S3} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix},$$

$$W^{C'}_{S4} = \begin{bmatrix} 1 & 2 & 0 & 2 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, W^{C'}_{S5} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, W^{C'}_{S6} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

With the new delay matrix, we can now recompute the structural robustness metric according to Equation 8.2. The numerator of the fractional part is computed as 16 units. The structural robustness metric

is computed as $1 - 16/40 = 0.6$, suggesting that this is a good move to perform to improve the structural robustness of the system.

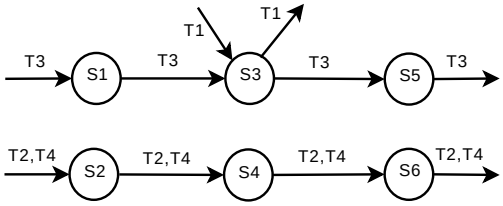

Figure 8.6: Configuration C'' after moving task $T1$ from $S4$ to $S3$

Next, let us move task $T1$ from instance $S4$ to $S3$, and let the new configuration be denoted as C'' as shown in Figure 8.6. All the row and column entries for task $T1$ on $S4$ are set to zero as $T1$ no longer executes on $S4$. As $T1$ executed only on $S4$, for each lower priority task, the segment of $T1$ that interfered with it consisted of only one stage, and these segments are now removed. At $S3$, $T1$ delays task $T3$ and the term $W^{C''}_{T1,S3,T3}$ is set to 2 units. Finally, the stage-additive component for task $T1$ is set as $W^{C''}_{T1,S3,T1} = 1$. The updated matrices for each instance for the new configuration C'' are computed as follows:

$$W^{C''}_{S1} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, W^{C''}_{S2} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, W^{C''}_{S3} = \begin{bmatrix} 1 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix},$$

$$W^{C''}_{S4} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, W^{C''}_{S5} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, W^{C''}_{S6} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Notice that, the numerator of the fractional part of the structural robustness metric is now further reduced to 14 units. The structural robustness of the system is increased to $1 - 14/40 = 0.65$. This was because we moved a higher priority task from one instance to another, such that the number of lower priority tasks that are affected is reduced, as illustrated in Figure 8.1(b).

Now that we know how to update the structural robustness metric when we change configurations, we need efficient ways to explore the space of all configurations to determine those that are more robust. We adopt a simple hill climbing algorithm that works as follows. We start with a random configuration and arbitrarily pick a task and resource instance and move it to another arbitrary instance of the same resource. If the metric for the new configuration is found to be higher than that for the current configuration, then we retain the new configuration. Otherwise, we discard it and try a new arbitrary change in configuration. Thus, the hill climbing algorithm will always proceed towards configurations that improve the value of the

structural robustness metric. The hill climbing algorithm can be easily modified to allow a limited number of steps that decrease the metric. This will allow the algorithm to explore a larger portion of the search space and to step out of local maxima.

8.4.2 Handling Tasks with Cyclic Paths

In this section, we extend our technique to tasks that may have cyclic paths, similar to our work in Chapter 5. When the path of a job T_k revisits a resource instance more than once, we say that it contains one or more *folds*. A fold of T_k starting at instance j is defined as the longest sequence of stages (in the order traversed by T_k) that does not repeat a resource instance twice. The first fold on $Path_k^C$ starts with the first resource that T_k visits. If the path of a task is acyclic, then it has only one fold that contains the whole path. We shall assume that each fold of a task is assumed independent of one another, and will be treated as separate higher priority jobs. The intuition behind defining folds is that each fold may delay a lower priority job at most once per stage.

Similar to our earlier definition, we can define task segments for each fold. The delay composition theorem that bounds the worst-case end-to-end delays of tasks is valid for tasks with cyclic paths as well. Each segment of each fold of a higher priority job causes a delay of at most two maximum stage execution times on a lower priority job. The rest of our discussion on improving the structural robustness of systems follows as before, using the delay composition theorem for tasks that may contain cyclic paths.

8.5 Evaluation

In this section, we evaluate through simulations the structural robustness of the system to unanticipated delays in the execution times of tasks. We first measure the number of deadline misses in the system in the absence of unanticipated delays. We then introduce unanticipated delays in the execution times by varying the fraction of sub-tasks that are delayed, as well as the extent to which they are delayed. We measure the end-to-end deadline misses before and after applying our algorithm to improve the robustness of the system, and show that the algorithm is able to reduce the number of deadline misses by around 50%.

The default system consists of 8 resource types and three instances of each resource. The system is assumed to operate close to the capacity. This is ensured by admitting enough tasks into the system, such that very few deadlines are missed in the absence of unanticipated delays in the worst-case stage execution times. Task routes are chosen by first choosing a path length at random, and then randomly picking a resource for each hop. Task routes can have cycles. Based on the sequence of resources for each task, we assign particular instances of resources to determine the task's path within the system. Each resource

instance serves tasks using a deadline monotonic scheduling policy. Other simulation parameters are chosen similar to previous chapters. The default value of the deadline ratio parameter, DR, is assumed to be 1.0. The default value of the task resolution parameter, T, is chosen as 0.1. The execution time at each stage is chosen within a range of 10% on either side of the mean.

Unexpected delays are introduced into the system, by picking a certain fraction of the sub-tasks, denoted as $DelayedTasks$, to experience unanticipated delays. The default value of $DelayedTasks$ is 0.25. The delay experienced by each sub-task thus chosen, is also varied as a parameter $DelayAmt$. The parameter $DelayAmt$ represents the ratio of the unanticipated delay to the original estimate of the worst-case execution time of the sub-task. Unless otherwise specified, the value of $DelayAmt$ is taken as 0.75.We consider two importance vectors for the tasks. The first assigns equal importance to all the tasks and the second assigns an importance to each task inversely proportional to its end-to-end deadline. As the results from both importance vectors are similar, we only plot the values for the case where the importance of all tasks are equal. We run our hill climbing algorithm for 500 steps, at each step picking a task to move from one instance to another, and retaining the new configuration if its structural robustness is found to be better than that of the current configuration.

Each point in the figures below represent the average value of 100 independent executions, with each execution consisting of 80000 task invocations (of all tasks taken together). The 95% confidence interval for all the values presented are within 1% of the mean value, and is not plotted for the sake of legibility.

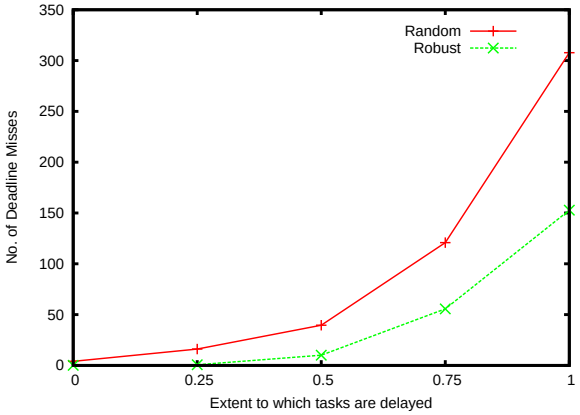

Figure 8.7: Comparison of number of deadline misses for different extents to which tasks are delayed

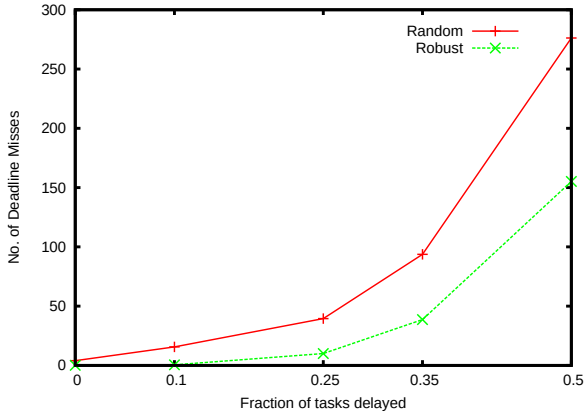

Figure 8.8: Comparison of number of deadline misses for different fraction of tasks delayed

We first varied the $DelayAmt$ for each unanticipated delay and estimated the number of deadline misses before applying are algorithm to improve robustness (labeled as *random*) and after (labeled as *robust*). The results are plotted in Figure 8.7. Note that a value of zero for $DelayAmt$ represents the system without any unanticipated delays. As expected, the number of deadline misses experienced by the baseline randomized

system increases with the value of $DelayAmt$. As the value of $DelayAmt$ is increased from 0 to 1, the number of deadline misses of the baseline system increases from almost zero to about 300. For each value of $DelayAmt$, applying our robustness algorithm to the task paths reduces the number of deadline misses by over 50%. This can be extremely useful to improve the overall performance of soft real-time systems, where estimations of worst-case computation times can be erroneous.

We next varied the fraction of tasks that experience unanticipated delays by varying the parameter $DelayedTasks$ from 0 to 0.5 and measured the number of deadline misses. The results of this experiment are plotted in Figure 8.8. Here again, a value of zero for the $DelayedTasks$ parameter denotes a system without unanticipated delays. The robustness algorithm is able to achieve more than a 60% reduction in the number of deadline misses for all values of $DelayedTasks$ up to 0.35, and achieves around 40% reduction when $DelayedTasks$ is 0.5.

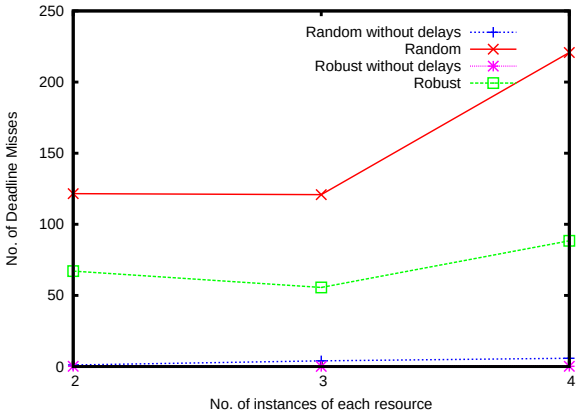

Figure 8.9: Comparison of number of deadline misses for different number of instances for each resource type

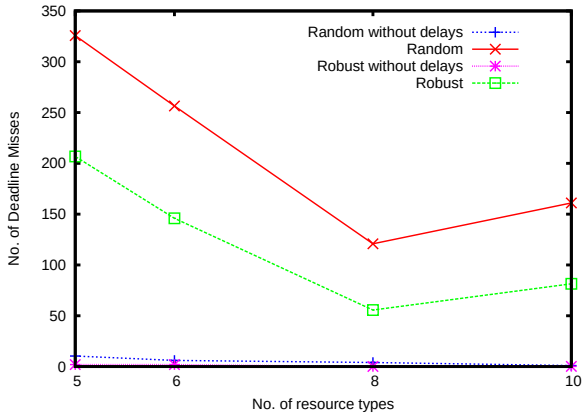

Figure 8.10: Comparison of number of deadline misses for different number of resource types

Figure 8.9 plots the number of deadline misses for different number of instances available for each resource type (there are 8 types of resources). For each system, the number of tasks admitted is varied to ensure that the system operates close to its capacity when there are no unanticipated delays. This is ensured by admitting as many tasks to cause very few deadline misses (less than 10 for each execution). The label 'random without delays' represents the average number of deadline misses for the baseline system. The label 'robust without delays' represents the same when the robustness algorithm is applied. Unanticipated delays are introduced into the system with 25% of the sub-tasks experiencing delay ($DelayedTasks = 0.25$) and each such sub-task being delayed for 75% additional time ($DelayAmt = 0.75$). The labels 'random' and 'robust' denote the number of deadline misses in the system with unanticipated delays before and after applying our robustness algorithm. We are able to achieve a 40-60% reduction in the number of deadline misses, with the reduction being larger for systems with more number of instances of each resource. This

is because, as more instances are available, the algorithm is able to perform better by finding more robust assignments of sub-tasks to stages.

Figure 8.10 plots the number of deadline misses for different number of types of resources. Similar to the previous experiment, the number of admitted tasks is varied to admit as many tasks as possible without exceeding 10 deadline misses for each execution of the system. The value of $DelayedTasks$ is set as 0.25 and that of $DelayAmt$ is set to 0.75. Here again the robustness algorithm is able to achieve a 35-50% reduction in the number of deadline misses, with the reduction being more pronounced for larger systems, as the algorithm has a greater potential to find better assignments of sub-tasks to stages.

Chapter 9

Application to Wireless Networks

The theory developed in this thesis can be applied to a wide range of application scenarios. It applies to any distributed system of resources that are scheduled in a prioritized manner. An important class of systems where the theory can be applied is in wireless networks. Wireless networks are becoming ubiquitous, ranging from mission-critical multi-hop ad hoc networks to urban mesh and personal networks. A large volume of the load carried by these networks are audio and video traffic with real-time requirements. In this chapter, we describe two extensions of our theory to the domain of wireless networks. The first, described in Section 9.1, applies to bandwidth allocation of real-time flows in wireless networks so as to maximize a notion of network utility in the presence of delay and capacity constraints. The second, described in Section 9.2, derives a scheduling mechanism that provides low and bounded end-to-end delay guarantees for packets of flows in a wireless network with arbitrary topology and arbitrary interference constraints.

9.1 Bandwidth Allocation for Elastic Real-Time Flows in Multi-hop Wireless Networks Based on Network Utility Maximization

We consider mission-critical cyber-physical wireless communication networks. These networks are dominated by audio and video traffic (e.g., voice communication among members and remote surveillance data from camera sensors). It is impossible to guarantee meeting all flow deadlines because of the unpredictable demand, dynamically changing network topology, variable levels of interference, and lack of strict priority-based resource allocation, resulting in collisions and out of order transmissions. Given these limitations, the goal of supporting real-time flows is approached by casting the problem as one of utility maximization, where utility depends on meeting deadlines. This problem then becomes a generalization of schedulability maximizing resource allocation, in which we seek an allocation of network resources to flows such that the most utility is achieved from meeting flow deadlines. Resource allocation is indirectly achieved by controlling flow rates while maintaining resource constraints. Fortunately, multimedia flows are especially amenable to

rate adaptation, which makes rate control mechanisms meaningful in this environment.

We formulate the problem as one of constrained Network Utility Maximization (NUM) of prioritized elastic flows, where priorities are set according to the delay constraints of flows. We assume that there is no utility in delivering a data item after its delay constraint is violated. When these constraints are met (or if no delay constraints are specified), the utility depends on the rate of the flow in question.

We adopt the worst-case delay bound obtained for Directed Acyclic Graphs [42] to derive a worst-case bound on the end-to-end delay of prioritized flows as a function of link flow rates. We then show how the constraints on delay and link capacity can be expressed purely based on variables that are known locally in the neighborhood of each node. This is done by defining a new variable at each hop i, to denote the ratio of the delay starting from the i^{th} hop and including all future hops to the deadline of the flow. Neighboring nodes periodically exchange values of all variables that are maintained. Therefore, the variable at the i^{th} hop is updated based on the value obtained from node at the $(i+1)^{st}$ hop and the estimate of the delay at the i^{th} hop.

We formulate a NUM problem using utilities of flows defined as concave functions of the flow rate, which has been a popular assumption in previous literature [59]. The solution to the NUM problem results in a distributed rate allocation algorithm which can be executed independently at each node. The algorithm converges to a rate allocation that maximizes utility, and at the same time guarantees that all flows meet their delay requirements. Results from simulations demonstrate that the algorithm is able to achieve a lower deadline miss ratio and a higher utility, in the presence of real-time traffic, compared to a rate control algorithm based on the traditional NUM formulation without delay constraints [59]. Further, we show that using the utility function, it is possible to differentiate between a flow's urgency (the priority is set according to the flow's urgency) and its importance in terms of the fraction of bandwidth requested. Therefore, it is possible to have short high-urgency flows, and prioritized treatment of such flows does not adversely affect important high-bandwidth non-realtime flows.

The rest of this section is organized as follows. In Section 9.1.1, we describe the system model, the problem, and notations used. The NUM problem formulation is presented in Section 9.1.2. A decentralized solution to the NUM and the resulting distributed algorithm are presented in Section 9.1.3. Issues in implementing the algorithm in a wireless network are discussed in Section 9.1.4. Section 9.1.5 presents results from simulations.

9.1.1 System Model and Problem Description

Consider a snapshot in time in the life of a cyber-physical multihop wireless network of nodes (such as soldiers and sensors) with wireless communication links, forming a particular topology. Each link l has an average capacity of ζ_l units that is roughly constant at the time scales of algorithm convergence. The links are to carry a possibly dynamic set of elastic end-to-end flows S. Each flow $s \in S$ is characterized by a path $Path_s$, from its source to its destination (as determined by the routing layer), and, optionally, an end-to-end latency requirement D_s, within which packets of the flow need to reach the destination. The subset of flows with latency requirements is denoted by the set S'. Each flow s also has a utility which is a concave function, f_s of its flow rate, x_s. Flows are assigned fixed priorities, and fixed priority scheduling is assumed at each intermediate node. Although the theory derived in this chapter is independent of how priorities are assigned, it would be prudent to assign priorities such that flows with tighter latency requirements have a higher priority during scheduling. Flows with no latency requirements are served at the lowest priority. In Section 9.1.4, we discuss how prioritized scheduling can be achieved in a distributed wireless scenario. In scenarios where nodes are mobile, we expect the algorithm presented in this chapter to still work, if the time frame at which routes change is much larger than the iteration interval of the distributed algorithm.

The objective of this formulation is to identify a distributed algorithm that can maximize a global (given) utility function, while still operating within the feasible region defined by capacity and delay constraints. Formulating and solving this as a NUM problem helps achieve this objective, and identifies the algorithm that sources and intermediate nodes should execute. Table 9.1 presents the notation that will be used in the rest of this chapter. We define $CC^i_{s',s}$, the contention count of flow s' at the i^{th} hop of flow s, as the number of transmission hops of flow s' that interfere with the i^{th} hop transmission of flow s.

C_i	Maximum time taken to process and forward a packet of flow i
N_s	Number of nodes in the route of flow s
D_s	Latency requirement of flow s
x_s	Rate of flow s
$\vec{x}$	$(x_s, s \in S)$, other vectors defined likewise
f_s	Utility function of flow s
$Path_s$	Path followed by flow s
$SM_{i,j}$	Number of times flow i's route splits from and then merges onto flow j's route.
ζ_l	Capacity of link l

Table 9.1: Notations used

9.1.2 Problem Formulation Based on Network Utility Maximization

In this section, we first derive the delay constraints on the rates of flows in the network, which are dependent on global information about the network. We show how decentralized capacity and delay constraints can be derived, each of which is based on variables that are local to a single node. This eliminates the need to maintain any global information. Note that delay constraints only apply to flows in set S' which have end-to-end delay requirements, whereas capacity constraints involve all the flows in the network, namely the set S. These decentralized constraints are then used in the NUM formulation, the solution of which leads to a distributed rate control algorithm.

Deriving the Delay Constraints

We adapt our delay composition result to wireless networks to obtain an end-to-end delay bound of a flow, in terms of the rates of other concurrent flows in the network. For convenience, we reproduce the non-preemptive delay composition theorem from Chapter 4 here. The theorem is targeted towards a multi-stage distributed system that processes several classes of real-time tasks. Each task requires processing at a sequence of resource stages, which forms a path in the distributed system. The theorem assumes deterministic knowledge of the execution parameters of all jobs that execute concurrently in the system. However, unlike queuing theory or network calculus, it does not make any assumptions on the arrival pattern (or distribution) of jobs, and computes the delay bound for a job assuming a worst-case arrival pattern for the given set of concurrent jobs. In deriving the worst-case end-to-end delay bound, it is shown that the delay that a higher priority job (a task invocation) causes a lower priority job is dependent on the number of times the paths of the two jobs split and merge with one another. The net end-to-end delay of a job is expressed as a sum of the delay due to interference from higher priority jobs and the delay incurred due to path length.

The theorem assumes the following notation: $C_{i,j}$ is the computation time of job J_i at stage j, $C_{i,max}$ is the maximum computation time of job J_i over all stages, $\bar{S}$ is the given set of jobs with higher priority than job J_1, $SM_{i,1}$ is the number of times the path of job J_i splits from and then merges onto the path of job J_1, and $M_1(j)$ is the set of jobs with lower priority than job J_1 whose paths merge with the path of J_1 at stage j. Based on the non-preemptive delay composition theorem, the end-to-end delay of job J_1 executing on N stages can be obtained as,

$$Delay(J_1) \leq \sum_{i \in \bar{S}} C_{i,max}(1 + SM_{i,1}) + \sum_{\substack{j \in Path_1 \\ j \leq N-1}} \max_{i \in S}(C_{i,j}) + \sum_{j \in Path_1} \max_{i \in M_1(j)} C_{i,max} \tag{9.1}$$

Informally, the first term in the delay bound can be thought of as the delay due to interference from

higher priority jobs, the second term is a hop penalty for each hop traversed by the job, and the third term is a blocking penalty due to lower priority jobs. While the theorem assumes prioritized scheduling, which is impossible to achieve exactly in wireless networks, solutions such as the emerging 802.11e standard [1] for the Enhanced Distributed Coordination Function (EDCF), have been developed to support approximate prioritized scheduling. More effective solutions to the prioritized scheduling problem at the MAC layer would result in better performance of our proposed algorithm. We discuss this problem further in Section 9.1.4 and describe the solution we adopted.

Let L be the size of a maximum sized packet. The time taken to process such a packet at a link j is $\frac{L}{\zeta_j}$. We can now derive the delay constraint for each flow s from the delay composition theorem as follows ($k < s$ denotes that flow k has a higher priority than flow s):

$$\sum_{\substack{Packets\ of \\ flow\ k,\ k \leq s}} C_k(1 + SM_{k,s}) + \sum_{j<N_s} \frac{L}{\zeta_j} + \sum_{j<N_s} \max_{i \in M_s(j)} C_i \leq D_s \tag{9.2}$$

The blocking penalty (the third term) can be considered to be negligible as it is at most the processing time of one lower priority packet at each hop (the higher priority packet will be transmitted ahead of the next lower priority packet). $(1 + SM_{k,s})$ denotes the number of times flow k merges onto the route of flow s. Inequality 9.2 can be rewritten as,

$$\sum_{j \leq N_s} \sum_{\substack{Packets\ of\ flows\ k \\ merging\ with\ flow\ s \\ at\ stage\ j,\ k \leq s}} C_{k,j} + \sum_{j<N_s} \frac{L}{\zeta_j} \leq D_s \tag{9.3}$$

where $C_{k,j}$ is the interference that a packet of flow k causes a packet of flow s at hop j. In a fully schedulable system, each packet of flow s is present in the network for at most D_s time units. A packet of flow s can encounter packets of flow k that arrived to the system D_k units before it, as well as packets that arrived D_s units after it arrived to the system (for flows that do not have end-to-end delay requirements, D_k can be the time-to-live for the packet in the network). It can therefore encounter packets of flow k whose inception was within a duration of $D_s + D_k$ time units. Further, at the j^{th} hop of flow s, $CC^j_{k,s}$ transmission hops of flow k interfere with flow s. Let us define additional variables $X_{L^i_s}$, denoting the forwarding rate of flow s at hop i, with $X_{L^0_s} = x_s$ (L^i_s is the link carrying the i^{th} hop of flow s). Then, the total delay at hop j due to all packets of flow k whose inception was within a duration of $D_s + D_k$ time units is $C_{k,j} = (D_s + D_k)\frac{X_{L^i_k} CC^j_{k,s}}{\zeta_{L^i_k}}$. Constraint 9.3 can now be written as,

$$\Big(\sum_{j \leq N_s} \sum_{\substack{Flows\ k\ at\ hop\ i \\ merging\ with\ flow\ s \\ at\ hop\ j,\ k \leq s}} \frac{X_{L^i_k} CC^j_{k,s}}{\zeta_{L^i_k}} (1 + \frac{D_k}{D_s}) \Big) + \frac{1}{D_s} \sum_{j<N_s} \frac{L}{\zeta_j} \leq 1 \tag{9.4}$$

Deriving Localized Capacity and Delay Constraints

In order to decentralize the solution and design a distributed algorithm, all constraints need to be expressed in terms of local variables only. Let tuple (s, i) denote the i^{th} hop of flow s. Let $Q(l)$ denote the set of tuples (s, i) that pass through the neighborhood of link l, that is, interfere with transmissions on link l. Note that in $Q(l)$, each flow may be represented multiple times based on the number of hops of flow s that interfere with transmissions along link l. Conversely, let $\bar{Q}(s, i)$ denote the set of links with which the i^{th} hop of flow s interferes. The capacity constraints can now be written as:

$$\sum_{(s,i)\in Q(l)} X_{L_s^i} \leq \zeta_l, \quad \forall\ l \in L$$

Further, to ensure that the output rate at each hop is at least as large as the input rate,

$$X_{L_s^i} \leq X_{L_s^{i+1}}, \quad \forall\ i, s \in S$$

In order to localize the delay constraint (9.4), we define additional variables $Y_{L_s^i}$, denoting the sum of the terms in constraint 9.4, starting from the i^{th} hop and including all future hops of flow s. $Y_{L_s^i}$ represents the ratio of the sum of the delays on all hops starting from the i^{th} hop, to the deadline of the flow. The delay constraint now becomes,

$$Y_{L_s^0} \leq 1, \quad \forall\ s \in S'$$

Let L_s^i be the link $l = (e, f)$, and let L_s^{i-1} be the link (d, e) (for $i > 0$). The following constraint governs $Y_{L_s^i}$ for all (s, i),

$$Y_{L_s^i} \geq Y_{L_s^{i+1}} + \frac{1}{D_s}\frac{L}{\zeta_l} + \sum_{\substack{s', i':\ s'<s;\\ L_{s'}^{i'}=(x,e), x\neq d}} \frac{X_{L_{s'}^{i'+1}} CC_{s',s}^{i}}{\zeta_{L_{s'}^{i'+1}}}\left(1 + \frac{D_{s'}}{D_s}\right) + \sum_{\substack{s':\ s'\leq s;\\ L_{s'}^{0}=(e,x), \forall\ x}} \frac{X_{L_{s'}^{0}} CC_{s',s}^{i}}{\zeta_{L_{s'}^{0}}}\left(1 + \frac{D_{s'}}{D_s}\right)$$

The summations on the RHS of the previous constraint adds up all the flow rates of higher priority flows that merge with flow s at the i^{th} hop, or have their source at the i^{th} hop of flow s. Let $l' = (g, h)$ be the last link of flow s. $Y_{L_s^{N_s}}$ should be at least as large as the sum of terms due to higher priority flows that merge with flow s at its destination. That is,

$$Y_{L_s^{N_s}} \geq \sum_{\substack{s', i':\ s'<s;\\ L_{s'}^{i'}=(x,h), x\neq g}} \frac{X_{L_{s'}^{i'+1}} CC_{s',s}^{N_s}}{\zeta_{L_{s'}^{i'+1}}}\left(1 + \frac{D_{s'}}{D_s}\right) + \sum_{\substack{s':\ s'<s;\\ L_{s'}^{0}=(h,x), \forall\ x}} \frac{X_{L_{s'}^{0}} CC_{s',s}^{N_s}}{\zeta_{L_{s'}^{0}}}\left(1 + \frac{D_{s'}}{D_s}\right)$$

NUM Formulation

The constraints derived above define a feasible region within which the network should operate in order to ensure that packets of flows do not exceed their latency requirements or are dropped due to lack of bandwidth in the network. To identify the utility maximizing point within this feasible region we can formulate the NUM with completely local constraints as,

$$\texttt{Maximize} \quad \sum_{s \in S} f_s(x_s), \quad \textit{subject to}$$

$$\sum_{(s,i) \in Q(l)} X_{L_s^i} \leq \zeta_l, \quad \forall\ l \in L \tag{9.5}$$

$$X_{L_s^i} \leq X_{L_s^{i+1}}, \quad \forall\ i, s \in S \tag{9.6}$$

$$Y_{L_s^0} \leq 1, \quad \forall\ s \in S' \tag{9.7}$$

$$Y_{L_s^i} \geq Y_{L_s^{i+1}} + \frac{1}{D_s}\frac{L}{\zeta_l} + \sum_{\substack{s',i':\ s'<s;\\ L_{s'}^{i'}=(x,e),x\neq d}} \frac{X_{L_{s'}^{i'+1}} CC_{s',s}^{i}}{\zeta_{L_{s'}^{i'+1}}}(1+\frac{D_{s'}}{D_s}) + \sum_{\substack{s':\ s'\leq s;\\ L_{s'}^{0}=(e,x),\forall\ x}} \frac{X_{L_{s'}^{0}} CC_{s',s}^{i}}{\zeta_{L_{s'}^{0}}}(1+\frac{D_{s'}}{D_s}), \forall\ i, s \in S'; L_s^i = (e,f) \tag{9.8}$$

$$Y_{L_s^{N_s}} \geq \sum_{\substack{s',i':\ s'<s;\\ L_{s'}^{i'}=(x,h),x\neq g}} \frac{X_{L_{s'}^{i'+1}} CC_{s',s}^{N_s}}{\zeta_{L_{s'}^{i'+1}}}(1+\frac{D_{s'}}{D_s}) + \sum_{\substack{s':\ s'<s;\\ L_{s'}^{0}=(h,x),\forall\ x}} \frac{X_{L_{s'}^{0}} CC_{s',s}^{N_s}}{\zeta_{L_{s'}^{0}}}(1+\frac{D_{s'}}{D_s}), \forall\ s \in S'; L_s^{N_s-1} = (g,h) \tag{9.9}$$

Constraint 9.5 ensures that the capacity requirement around the interference neighborhood of any link is within the capacity of the link. Constraint 9.6 ensures the continuity of each flow, that is, the rate of each flow out of a node should be at least as large as the rate of flow into that node. Constraints 9.7, 9.8, and 9.9 jointly constitute the delay constraints. Constraint 9.7 ensures that the end-to-end delay is less than the latency requirement for each flow and is maintained at the source of the flow. Constraint 9.8 implemented by intermediate nodes in the route of any flow, maintains an estimate of the interference due to higher priority flows on all subsequent nodes in the route of this flow. Finally, Constraint 9.9 is implemented by the destination of each flow, and accounts for the interference due to flows at the destination. Thus, the interference due to higher priority flows is accumulated along the backward route from destination to source, so that the net end-to-end delay to latency requirement ratio ($Y_{L_s^0}$) can be estimated at the source.

9.1.3 Decentralized Solution and Distributed Algorithm

In this section, we solve the NUM problem using Lagrangian decomposition. A tutorial on decomposition methods for NUM can be found in [70]. We first construct the Lagrangian, and then differentiate the Lagrangian with respect to each of the variables to obtain the update equations for the respective variables. Finally, we present a distributed algorithm based on the derived update equations. The algorithm, when executed independently by each node, collectively assigns rates to flows to maximize global network utility, while ensuring that the latency requirements of all flows are met. The Lagrangian can now be defined as:

$$\mathbf{U} = \sum_{s\in S} f_s(x_s) + \sum_{l\in L}\lambda_l\Big(1-\frac{1}{\zeta_l}\sum_{(s,i)\in Q(l)} X_{L_s^i}\Big) + \sum_{s\in S}\sum_{i=0}^{N_s-1}\mu_{s,i}\Big(X_{L_s^{i+1}} - X_{L_s^i}\Big) + \sum_{s\in S'}\delta_s\Big(1-Y_{L_s^0}\Big)$$
$$+\sum_{s\in S'}\sum_{\substack{i=0,\\ L_s^{i-1}=(d,e)\\ l=L_s^i=(e,f)}}^{N_s-1}\gamma_{s,i}\Big(Y_{L_s^i} - Y_{L_s^{i+1}} - \frac{1}{D_s}\frac{L}{\zeta_l} - \sum_{\substack{s',i':\, s'<s;\\ L_{s'}^{i'}=(x,e),x\neq d}}\frac{X_{L_{s'}^{i'+1}}CC_{s',s}^{i}}{\zeta_{L_{s'}^{i'+1}}}(1+\frac{D_{s'}}{D_s}) - \sum_{\substack{s':\, s'\leq s;\\ L_{s'}^0=(e,x),\forall\ x}}\frac{X_{L_{s'}^0}CC_{s',s}^{i}}{\zeta_{L_{s'}^0}}(1+\frac{D_{s'}}{D_s})\Big)$$
$$+\sum_{s\in S'}\epsilon_s\Big(Y_{L_s^{N_s}} - \sum_{\substack{s',i':\, s'<s;\\ l'=(g,h)\\ L_{s'}^{i'}=(x,h),x\neq g}}\frac{X_{L_{s'}^{i'+1}}CC_{s',s}^{N_s}}{\zeta_{L_{s'}^{i'+1}}}(1+\frac{D_{s'}}{D_s}) - \sum_{\substack{s':\, s'<s;\\ L_{s'}^0=(h,x),\forall\ x}}\frac{X_{L_{s'}^0}CC_{s',s}^{N_s}}{\zeta_{L_{s'}^0}}(1+\frac{D_{s'}}{D_s})\Big)$$

Differentiating with respect to x_s and setting $\frac{\partial U}{\partial x_s} = 0$,

$$f_s'(x_s) = \sum_{l\in\bar{Q}(s,0)}\frac{\lambda_l}{\zeta_l} + \sum_{\substack{s',i':\, s'>s;\\ L_{s'}^{i'}=(source(s),x)}}\frac{\gamma_{s',i'}CC_{s,s'}^{i'}}{\zeta_{L_s^0}}(1+\frac{D_s}{D_{s'}}) + \frac{\gamma_{s,0}}{\zeta_{L_s^0}} + \mu_{s,0} + \sum_{\substack{s':\, s'>s;\\ dest(s')=source(s)}}\frac{\epsilon_{s'}CC_{s,s'}^{N_{s'}}}{\zeta_{L_s^0}}(1+\frac{D_s}{D_{s'}}) \tag{9.10}$$

In the above equation, the gammas and epsilons are summed over lower priority flows with which flow s interferes at its source (for flows that do not have delay constraints, δ, γ, and ϵ are assumed to be zero).

Differentiating with respect to $X_{L_s^i}, i > 0$, we obtain the update equation for $X_{L_s^i}$ using the gradient method [70] as,

$$X_{L_s^i}(t+1) = \Big[X_{L_s^i}(t) + \alpha_1(t)\Big(-\sum_{l\in\bar{Q}(s,i)}\lambda_l - \mu_{s,i} + \mu_{s,i-1}$$
$$-\sum_{\substack{s'>s, L_s^{i-1}=(x,e),\\ L_{s'}^{i'-1}=(d,e),x\neq d}}\frac{\gamma_{s',i'}CC_{s,s'}^{i'}}{\zeta_{L_s^i}}(1+\frac{D_s}{D_{s'}}) - \sum_{\substack{s'>s, L_s^{i-1}=(x,h),\\ L_{s'}^{N_{s'}-1}=(g,h),x\neq g}}\frac{\epsilon_{s'}CC_{s,s'}^{N_{s'}}}{\zeta_{L_s^i}}(1+\frac{D_s}{D_{s'}})\Big)\Big]^+ \tag{9.11}$$

In the above equation, the gammas and epsilons are summed over lower priority flows with which flow s merges (interferes) at its i^{th} hop.

Differentiating with respect to $Y_{L_s^0}$, the update equation for $Y_{L_s^0}$ is obtained as,

$$Y_{L_s^0}(t+1) = \Big[Y_{L_s^0}(t) + \alpha_2(t)\Big(- \delta_s + \gamma_{s,0}\Big)\Big]^+ \tag{9.12}$$

Differentiating with respect to $Y_{L_s^i}, N_s > i > 0$,

$$Y_{L_s^i}(t+1) = \Big[Y_{L_s^i}(t) + \alpha_2(t)\Big(\gamma_{s,i} - \gamma_{s,i-1}\Big)\Big]^+ \tag{9.13}$$

Differentiating with respect to $Y_{L_s^{N_s}}$,

$$Y_{L_s^{N_s}}(t+1) = \Big[Y_{L_s^{N_s}}(t) + \alpha_2(t)\Big(\epsilon_s - \gamma_{s,N_s-1}\Big)\Big]^+ \tag{9.14}$$

Differentiating with respect to λ_l,

$$\lambda_l(t+1) = \Big[\lambda_l(t) - \alpha_3(t)\Big(1 - \frac{1}{\zeta_l}\sum_{(s,i)\in Q(l)} X_{L_s^i}\Big)\Big]^+ \tag{9.15}$$

The update equations for the other dual costs $(\vec{\mu}, \vec{\delta}, \vec{\gamma}, \vec{\epsilon})$ can similarly be written down directly from the respective constraints they represent as follows:

$$\mu_{s,i}(t+1) = \mu_{s,i}(t) - \alpha_4(t)\Big(X_{L_s^{i+1}} - X_{L_s^i}\Big) \tag{9.16}$$

$$\delta_s(t+1) = \Big[\delta_s(t) - \alpha_5(t)\Big(1 - Y_{L_s^0}\Big)\Big]^+ \tag{9.17}$$

$$\begin{aligned}\gamma_{s,i}(t+1) \;=\; \Big[\gamma_{s,i}(t) - \alpha_6(t)\Big(Y_{L_s^i} - Y_{L_s^{i+1}} - \frac{1}{D_s}\frac{L}{\zeta_l} \\ - \sum_{\substack{s',i':\, s'<s;\\ L_{s'}^{i'}=(x,e),x\neq d}} \frac{X_{L_{s'}^{i'+1}} CC_{s',s}^i}{\zeta_{L_{s'}^{i'+1}}}(1+\frac{D_{s'}}{D_s}) - \sum_{\substack{s':\, s'\leq s;\\ L_{s'}^0=(e,x),\forall\ x}} \frac{X_{L_{s'}^0} CC_{s',s}^i}{\zeta_{L_{s'}^0}}(1+\frac{D_{s'}}{D_s})\Big)\Big]^+\end{aligned} \tag{9.18}$$

$$\epsilon_s(t+1) = \Big[\epsilon_s(t) - \alpha_7(t)\Big(Y_{L_s^{N_s}} - \sum_{\substack{s',i':\, s'<s;\\ l'=(g,h)\\ L_{s'}^{i'}=(x,h),x\neq g}} \frac{X_{L_{s'}^{i'+1}} CC_{s',s}^{N_s}}{\zeta_{L_{s'}^{i'+1}}}(1+\frac{D_{s'}}{D_s}) + \sum_{\substack{s':\, s'<s;\\ L_{s'}^0=(h,x),\forall\ x}} \frac{X_{L_{s'}^0} CC_{s',s}^{N_s}}{\zeta_{L_{s'}^0}}(1+\frac{D_{s'}}{D_s})\Big)\Big]^+ \tag{9.19}$$

Note that the above update equations can be implemented by each node based purely upon information available to it from its neighbors. A node does not require any knowledge of flows outside its neighborhood. Based on these update equations, we obtain the following rate allocation algorithm:

Algorithm DeadlineAwareRateAllocation:

Initialize $\vec{X}$,$\vec{Y}$,$\vec{\lambda}$,$\vec{\mu}$,$\vec{\delta}$,$\vec{\gamma}$,$\vec{\epsilon}$

Repeat the following four steps

indefinitely:

1. Based on current values of X's,Y's,

update values for dual costs $\vec{\lambda}$,$\vec{\mu}$,$\vec{\delta}$,$\vec{\gamma}$,$\vec{\epsilon}$
using Equations 9.15, 9.16, 9.17, 9.18, and 9.19

2. Exchange values for the dual costs
with neighboring nodes
3. Recompute new values for X's and Y's
based on the current dual costs using
Equations 9.11, 9.12, 9.13, and 9.14
4. Update source rates using Equation 9.10
5. Exchange values for X's and Y's with
neighboring nodes

For the initialization step of the algorithm, all dual costs $\vec{\lambda}$, $\vec{\mu}$, $\vec{\delta}$, $\vec{\gamma}$, and $\vec{\epsilon}$ can be set to zero. X for each flow can be initialized to a constant flow rate at which all flows begin. Y for each hop of every flow can be initialized to the right hand side of Inequalities 9.7, 9.8, or 9.9 as applicable. Note that the periodic communication steps 2 and 5 of the algorithm can be executed asynchronously with the local computation steps 1, 3, and 4. The rate of sending updates can then be decreased to reduce the algorithm overhead. This would then cause nodes to use old values for the different variables. As the values for the different variables only change slightly during each iteration of the algorithm, reducing the rate of sending updates can significantly reduce overhead while not appreciably compromising performance. Also, updates can be piggy-backed on regular messages to reduce the need for update messages. The number of update messages required by the deadline aware rate allocation algorithm is the same as that of the rate allocation algorithm based on the traditional NUM formulation without delay constraints [59], and each update requires only a few extra bytes for the additional variables introduced by the delay constraints.

9.1.4 Implementation Considerations

In this section, we discuss several issues in implementing the algorithm described in Section 9.1.3. The non-preemptive delay composition theorem, used in deriving the delay constraint 9.4 assumes prioritized scheduling at each intermediate node. As exact prioritized scheduling is impossible to achieve in a distributed wireless network, we implement approximate prioritized scheduling as follows. Each node maintains independent queues for packets of each priority class and always chooses to transmit a packet of higher priority before transmitting a packet of lower priority. Further, in order to prioritize packet scheduling across neighboring nodes, we allow packets of higher priorities to have a smaller minimum contention window size compared to packets of lower priority. In our simulations, we support eight priority levels. This implemen-

tation is similar to the emerging 802.11e standard [1] for the Enhanced Distributed Coordination Function (EDCF), except that the queues do not act as virtual terminals, and packets from higher priority queues are always picked ahead of packets from lower priority queues. Prioritized scheduling can also be achieved using MAC protocols such as [30]. This problem has been addressed in past literature and is orthogonal to the problem we address. Better solutions to MAC layer prioritization will enhance the performance of our algorithm.

Note that the update equations for the algorithm require that nodes are aware of the dual costs and values for X's and Y's computed by nodes in their interference neighborhood, and not just the communication neighborhood (for example, the capacity constraint ensures that the sum of all flow rates in the interference neighborhood of each link is at most as large as the capacity of the link). This behooves the presence of a protocol at the network layer that can obtain this information. Identifying nodes that lie within interference range of a given node is a challenging problem that has been addressed in prior literature in wireless networks (such as [98]). We empirically measured the interference range to be 440m for each link when the communication range was 200m, and used this value in our simulations.

The NUM formulation assumes concave utility functions for the flows, and the optimization objective is to maximize the sum of utilities of all flows in the network. Utility functions serve as a measure of user satisfaction, and can also be used to control the trade-off between efficiency and fairness. For example, [65] defines a family of utility functions targeted at fairness. As the problem of choosing utility functions has been dealt with in literature, we keep our theoretical framework general and not dependent on any particular utility function. In our simulations, we consider two simple logarithmic utility functions to be used with the NUM formulation. In the first utility function, the utility is assumed to be proportional to priority:

$$f_s(x_s) = (Max\_Priority - W_s + 1) \log x_s, \tag{9.20}$$

where W_s is the priority of flow s (higher value implies lower priority), and $Max\_Priority$ is the maximum permissible priority value. Remember, however, that priority in our framework is set according to urgency, which in general, may not be proportional to utility. Hence, we also consider a utility function that is orthogonal to priority (i.e., urgency), where all flows have the same importance:

$$f_s(x_s) = \log x_s, \tag{9.21}$$

As mentioned earlier, the above utility functions are true only when delay constraints are met. If delay constraints are violated, the utility is zero. In practice, however, applications such as video streaming can tolerate some deadline misses. Hence, in the evaluation section, rather than dropping utility to zero abruptly,

we drop it linearly in the miss ratio as follows:

$$App\_Utility = f_s(x_s) * (1 - 10 * DMR), \quad DMR < 0.1$$
$$= 0, \quad otherwise \tag{9.22}$$

where $f_s(x_s)$ is defined as in Equations (9.20) or (9.21), and DMR is the deadline miss ratio for the flow. Note that the above application utility function reduces to $f_s(x_s)$, when no deadline misses occur. As the NUM-based algorithms operate within the feasible region where no deadline misses occur, the optimal solution also lies within this feasible region. At the optimal solution (and at all points within the feasible region), the value of the above application utility function would be the same as that of $f_s(x_s)$. Hence, the operation of the NUM-based algorithm remains the same regardless of how utility is defined outside the feasible region. We therefore use $f_s(x_s)$ defined as per Equations (9.20) or (9.21) as the utility function for the NUM-based algorithms, but use the application utility defined in Equation (9.22) as a metric of performance in our evaluations in Section 9.1.5.

Finally, the update equations are based on update parameters α. For the update equation for λ, we used a value of 0.5 ($\alpha_3 = 0.5$). For μ, we used a value of 0.2. For all the other update equations we used a value of 1.0. While we observed that changing these values affect the rate of convergence, for most values of the update parameters, the flow rates converged within 100 iterations of the algorithm. Convergence of a number of NUM problems has been studied in the past [15, 35]. Theoretically studying the convergence and stability properties of the algorithm will be a useful future work.

9.1.5 Simulation Results

In this section, we present results from simulations on ns2 [67]. Our default system consists of 50 nodes placed uniformly at random. We assume that nodes are stationary. The MAC layer protocol is assumed to be 802.11, with prioritized scheduling as described in Section 9.1.4. We consider a default of 5 elastic flows in the system, whose sources and destinations are chosen uniformly at random. The average hop length of the flows was between 3 and 4. The routing protocol is assumed to be DSDV. Link bandwidth is assumed to be 1 Mbps. One iteration of the rate allocation algorithm executes every 0.5 seconds. The flows are assumed to transmit at a constant bit rate chosen by the rate allocation algorithm. For traffic that is bursty, such as video, application-level buffers can be used to smoothen the burst and transmit at the average (roughly constant) rate. Prior theory such as [21] can be used to bound the delay due to buffering as a function of the original burstiness. While the end-to-end delay of a packet is the sum of this buffering delay and the network delay, in this work, we concern ourselves only with the network delay. The buffering delay can be

estimated as discussed in prior work [21].

In our evaluation, we compare the proposed deadline-aware rate allocation algorithm (which we call 'NUM with Delay Constraints'), with four other algorithms. The first is a rate allocation algorithm determined by the traditional NUM formulation without the delay constraints [59, 51], that maximizes utility in the presence of capacity constraints alone. This algorithm uses prioritized scheduling at the MAC layer and is referred as 'NUM without Delay Constraints'. In order to demonstrate that prioritized scheduling broadens the space of acceptable flow rates, we choose the second algorithm to be the same as the first except that scheduling at the MAC layer is FIFO. We call this 'NUM w/o Delay Constraints w/o Prioritized Scheduling'. For the third algorithm, we simulate prioritized scheduling at the MAC layer, in the absence of any rate control (called 'No Rate Control'). Here, each source of flow is assumed to transmit at the application-specified maximum packet generation rate. This serves as a baseline to analyze the advantages of applying rate control. Finally, to show that our algorithm can work with any prioritized MAC protocol, we evaluate the deadline-aware rate allocation algorithm with 802.11e as the MAC layer protocol (we call this 'NUM with Delay Constraints with 802.11e'). Table 9.2 shows the differences between the different algorithms. For the rate allocation algorithms based on the NUM formulations with and without the delay constraints, we evaluated both utility functions specified in Section 9.1.4 under Equations (9.20) and (9.21). The results when all flows have the same utility function (Equation (9.21)) are labeled with the suffix 'Eq. Util. Flows'. The results when the utilities of flows are directly proportional to their priority (Equation 9.20) do not have this suffix. The above algorithms are compared in terms of the achieved throughput and deadline miss ratio for each priority class.

Algorithm	Rate Control	Delay Constr.	Prior. Sched.
No rate control	No	No	Yes
NUM w/o delay	Yes	No	Yes
NUM w/o delay w/o prioritization	Yes	No	No
NUM with delay	Yes	Yes	Yes
NUM with delay,802.11e	Yes	Yes	802.11e

Table 9.2: The different algorithms being compared

All values presented are averaged over 50 simulation runs each lasting for 100 seconds. For each run, results were collected after a duration long enough to ensure that the rate control algorithm had stabilized. Y error bars indicate 95% confidence intervals.

Audio and video flows are typically governed by a maximum traffic generation rate. We imposed different maximum application traffic generation rates that would act as a ceiling for the transmission rates for each flow. We considered traffic generation values equal to 25, 50, 75, 100, 125, and 200 Kbps. A traffic generation

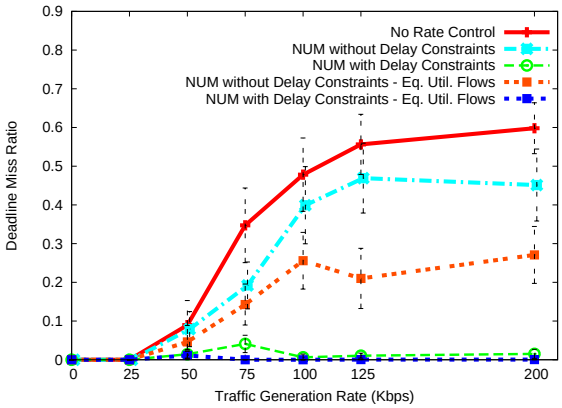

Figure 9.1: Deadline miss ratio, high priority flows

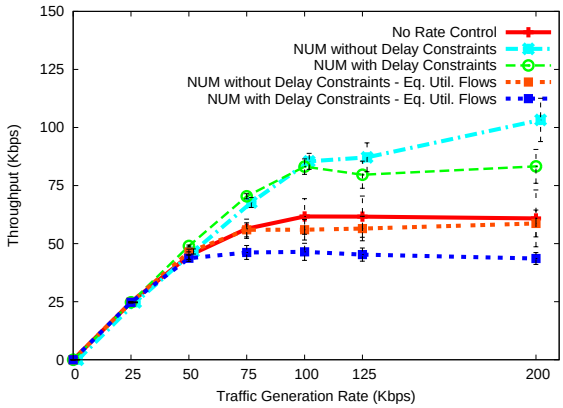

Figure 9.2: Throughput, high priority flows

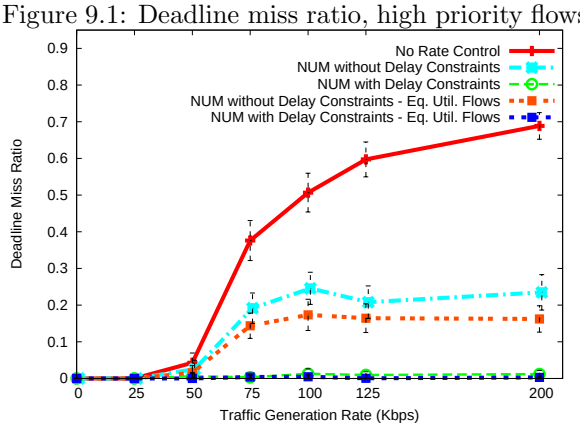

Figure 9.3: Deadline miss ratio, medium priority flows

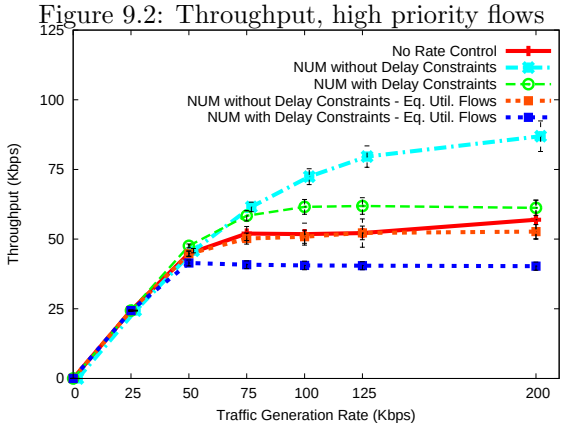

Figure 9.4: Throughput, medium priority flows

rate of 200 Kbps was observed to be nearly the same as not imposing any limit on the application traffic generation rate. We considered three priority levels for the flows, with end-to-end latency requirements of 2, 4, and 7 seconds. These values reflect typical requirements in military communications and hence were given specific attention. The number of flows of each priority was in the ratio 1:2:4, that is, there were four times as many low priority flows as there were high priority flows. The deadline miss ratios and average achieved throughput were measured for the different algorithms for each priority class. Figures 9.1 and 9.2 plot the comparison for high priority flows. Likewise, Figures 9.3 and 9.4 show the results for medium priority flows, and Figures 9.5 and 9.6 show the results for low priority flows. For all the three priority classes, the algorithm based on the NUM formulation with delay constraints, was able to ensure a deadline miss ratio of less than 5% of the packets, regardless of how the utility functions of flows are defined. In contrast, the algorithm based on the NUM formulation without delay constraints suffered a much higher deadline miss ratio for all priority classes especially at high load scenarios. The deadline miss ratio was observed to be the highest when no rate control was imposed. Further, for the NUM formulation without delay constraints, when prioritized scheduling was replaced by FIFO scheduling, more deadline misses were observed for medium

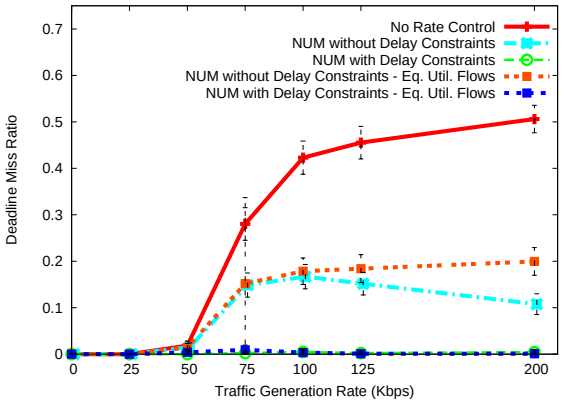

Figure 9.5: Deadline miss ratio, low priority flows

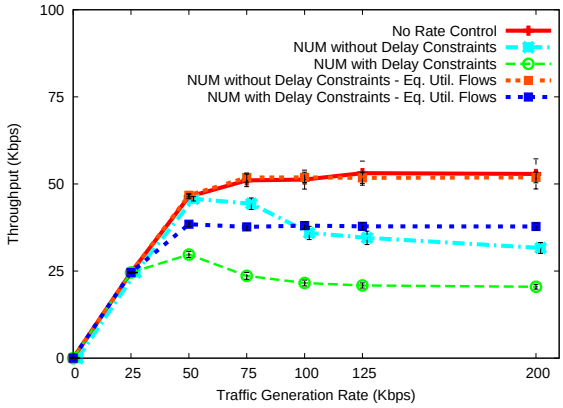

Figure 9.6: Throughput, low priority flows

and high priority flows, demonstrating the importance of prioritized scheduling. Finally, using 802.11e as the MAC layer protocol the deadline miss ratio and throughput values were found to be similar to when higher priority packets were always transmitted ahead of lower priority packets (instead of the virtual node concept of 802.11e), suggesting that our rate control algorithm would work well with any prioritized MAC protocol.

It can be observed from Figures 9.2, 9.4, and 9.6, that when the utilities of flows are proportional to their priority (for both the NUM with delay constraints and the NUM without delay constraints), the throughput of high priority flows is higher than that of low priority flows. However, when flows have the same utility function regardless of their priority, the throughput of all three priority classes are nearly equal. This conforms with theory suggesting that the throughput (or bandwidth share) that each flow obtains is only dependent on its importance as defined by the utility function, and is independent of the flow priorities. Thus, in a network with both real-time and non-real-time flows, the presence of delay constraints for some flows will not adversely affect the throughput of important non-real-time flows that may operate at the lowest priority. The utilities thus provide a mechanism to specify the importance of different flows in the network, and allocate bandwidth according to their importance.

The rate control algorithm based on the NUM with delay constraints is able to achieve a throughput within 20% of the throughput achieved by the NUM without delay constraints (for both utility function choices). This throughput penalty for imposing and ensuring that the latency constraints of flows are met is acceptable, as receiving a smooth video at low resolution is typically better than a high resolution video that often keeps freezing.

For the same experiment, we measured the application utility defined in Equation 9.22 as follows. In order to distinguish short bursty packet deadline misses and losses from prolonged poor performance, we measured the application utility for every 5 second interval and computed the average across all such intervals and

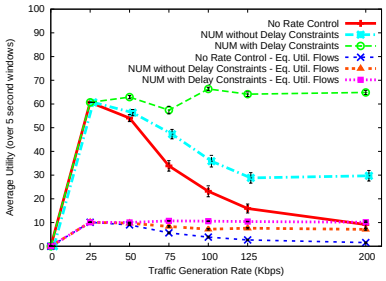

Figure 9.7: Average utility of high priority flows

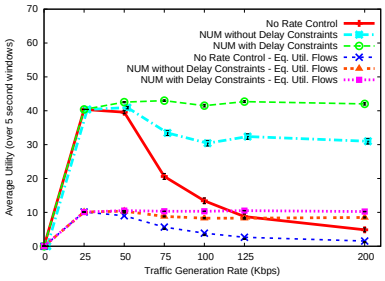

Figure 9.8: Average utility of medium priority flows

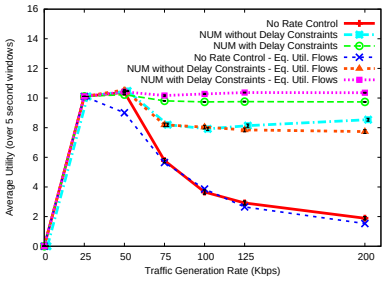

Figure 9.9: Average utility of low priority flows

simulation runs. This is presented for the different algorithms in Figures 9.7, 9.8 and 9.9, for high, medium, and low priority flows, respectively. For the deadline-aware rate control algorithm, the degradation in utility is only marginal with increase in traffic generation rate. In contrast, the other algorithms suffer a significant utility degradation for real-time flows except when the network is underloaded.

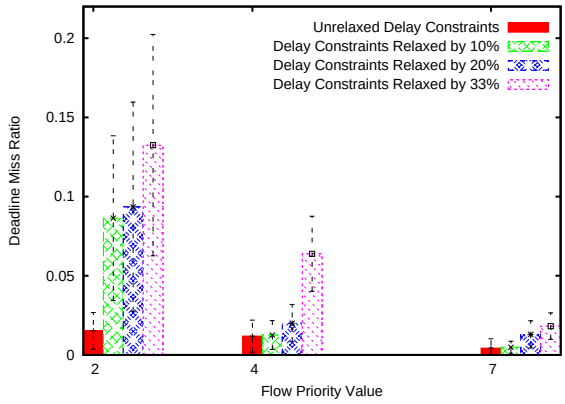

Figure 9.10: Deadline miss ratio achieved by the deadline-aware rate control algorithm when the delay constraint is relaxed

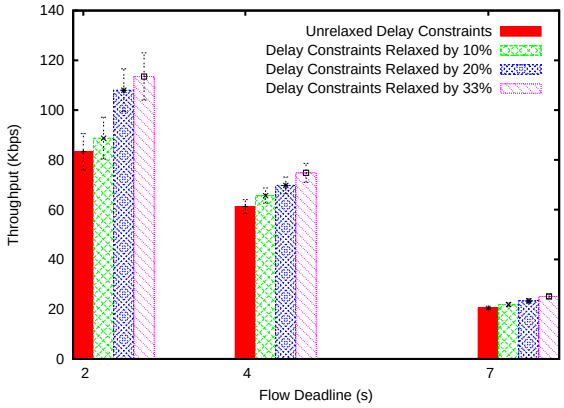

Figure 9.11: Throughput achieved by the deadline-aware rate control algorithm when the delay constraint is relaxed

In order to estimate how accurate or pessimistic the delay constraint is, we progressively relaxed the delay constraint and measured the deadline miss ratio and throughput for different priority classes for the algorithm based on the NUM formulation with delay constraints (shown in Figures 9.10 and 9.11). When the delay constraints were relaxed by 10%, the throughput increased by about 5%, but the deadline miss ratio nearly doubled. This shows that the delay constraints derived in this chapter are reasonably accurate.

In order to study the stability of the algorithm under dynamic load conditions, we conducted experiments where we allowed flows to enter and leave the network during the course of the experiment. Flows start with a transmission rate of 50 Kbps, and the deadline aware rate control algorithm is then used to adjust the transmission rate of flows. The transmission rates of flows and total utility (individual flow utilities were assumed to be proportional to their priority as defined in Equation (9.20)) are plotted versus time in Figure 9.12 for this experiment. At time 0, there were three flows in the network, flows 1, 2, and 3, with

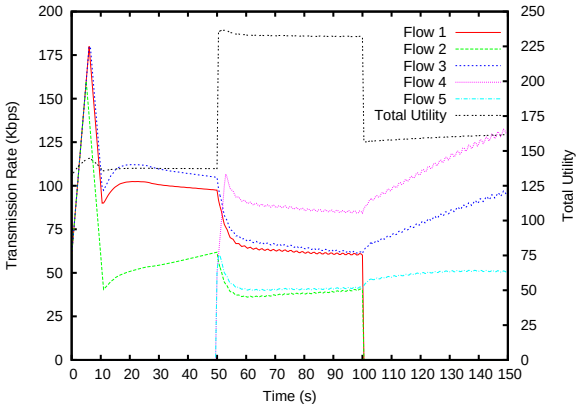

Figure 9.12: Transmission rate and total utility vs. time for a dynamic set of flows in the network

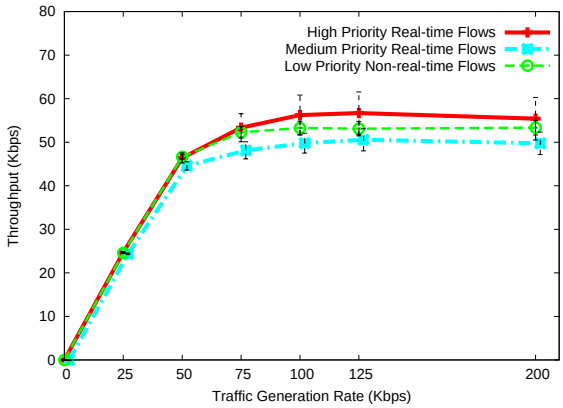

Figure 9.13: Transmission rate and total utility vs. time for a dynamic set of flows in the network

priority values 4, 7, and 4 (flow deadlines were same as the priority value in seconds). Fifty seconds into the experiment, flows 4 and 5, with priority values 2 and 7 were started. Note the drop in transmission rate for the first three flows and the increase in utility at time 50. At time 100, flows 1 and 2 leave the network. Note the increase in transmission rates for the other three flows at time 100. The algorithm was observed to stabilize within 10 seconds of each instance of flows entering or leaving the network.

In order to show that the NUM formulation in the presence of delay constraints does not discriminate against non-realtime flows, we conducted an experiment in the presence of both real-time and non-realtime flows. For this experiment, the lowest priority flows (with priority value 7) were made non-real-time flows. The high and medium priority flows were real-time flows with priority values 2 and 4 (equal to their respective deadlines). In order to demonstrate that the throughput allocated to each flow is only determined by its importance defined in the utility function, and not so much by its priority, all flows were assigned the same utility function. The throughput for each priority level for different traffic generation rates is plotted in Figure 9.13. The plot shows that all three priority levels receive nearly the same throughput regardless of their priority. Thus, the importance in the utility function can be adjusted to differentially allocate bandwidth to different flows regardless of whether they have delay constraints.

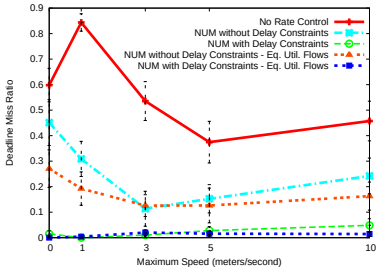

Figure 9.14: Deadline miss ratio of high priority flows for different mobility rates

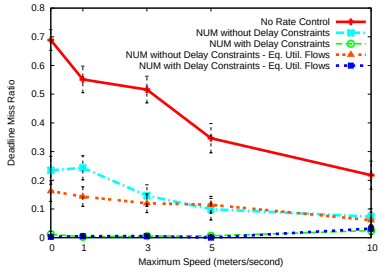

Figure 9.15: Deadline miss ratio of medium priority flows for different mobility rates

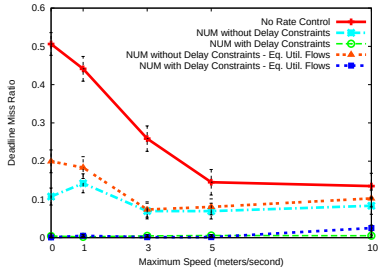

Figure 9.16: Deadline miss ratio of low priority flows for different mobility rates

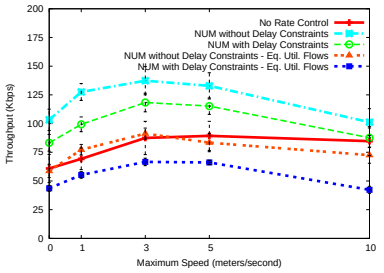

Figure 9.17: Throughput received by high priority flows for different mobility rates

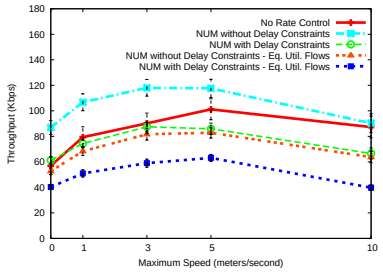

Figure 9.18: Throughput received by medium priority flows for different mobility rates

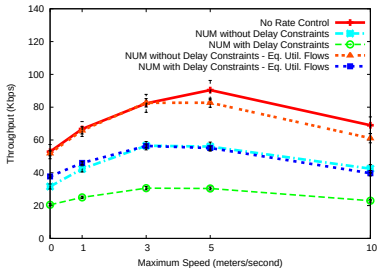

Figure 9.19: Throughput received by low priority flows for different mobility rates

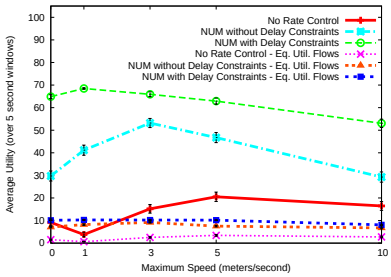

Figure 9.20: Average utility of high priority flows

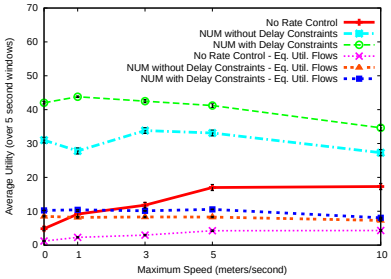

Figure 9.21: Average utility of medium priority flows

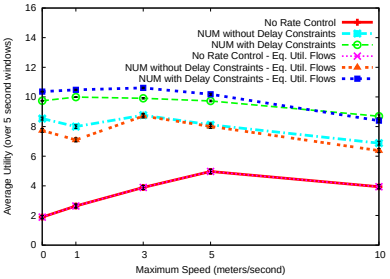

Figure 9.22: Average utility of low priority flows

We next evaluate the performance of the different algorithms when nodes are mobile, which is a typical scenario for mission critical cyber-physical wireless networks of nodes. We used a simple mobility model, wherein each node picks a destination at random, and move towards it at a uniform random speed (limited by a specified maximum speed). Upon reaching the destination, the node will pick another random destination and repeat this loop forever. We considered maximum speed values of 1,3,5, and 10 m/s. These values reflect speeds ranging from the typical walking speed of people to the speed of fast moving vehicles. The deadline miss ratios of the various algorithms for this experiment are shown in Figures 9.14, 9.15, and 9.16, for the three priority classes, respectively. The average achieved throughput is shown in Figures 9.17, 9.18, and 9.19. The total utility over all the flows for each priority class is plotted in Figures 9.20, 9.21, and 9.22 (the utility was computed over 5 second intervals, as described before).

At low mobility values (1 and 3 m/s maximum speeds) the throughput of the algorithms were found to actually increase. A possible reason is that mobility causes any congestion in the network to clear up and congestion never lasts for a long time. Since communication radius is about 200 m, it takes a significant amount of time (longer than the time required for algorithm convergence) for nodes to move out of range and the topology to change. At higher mobility values, there are a lot of packet losses causing the throughput to drop. Also the deadline miss ratio increases for high mobility values causing the utility to drop.

9.2 Minimizing End-to-End Delay in Wireless Networks Using a Coordinated EDF Schedule

In this section, we consider the problem of scheduling data in wireless networks such that the end-to-end delay is small. There has been a significant amount of work in recent years looking at scheduling a network such that *throughput optimality* is maintained, i.e., to ensure that the network always supports any set of flow rates that lie in the schedulable region. A common tool for this problem is the well-known backpressure protocol of Tassiulas and Ephremides [86]. However, much less attention has been paid to the problem of analyzing the end-to-end delay of a distributed wireless schedule.

Our approach to the delay minimization problem combines results from delay-based scheduling in wireline networks with results for throughput maximization in wireless networks. In particular, consider a set of flows passing through a wireline link. For simplicity, we shall focus on the case in which time is slotted and each link can transmit one packet per slot. We assume a leaky bucket model, with burst parameter for flow i assumed to be σ_i and the rate parameter to be ρ_i, i.e., the number of packets that arrive for flow i in a time interval of length t is at most $\sigma_i + \rho_i t$. We focus our attention on flow-based schedulers, since non-flow based schedulers such as FIFO are not throughput optimal [5] and do not provide protection for a flow from misbehaving flows. One of the most commonly studied flow-based protocols is the Weighted Fair Queuing (WFQ) protocol, that maintains a virtual Generalized Processor Sharing (GPS) protocol in the background to divide service among the flows according to the flow rates. WFQ always serves the packet that would finish earliest under the GPS protocol, assuming that no more packets arrive. Parekh and Gallager [71, 2] showed that the end-to-end delay for WFQ has the form $\frac{\sigma_i+K_i}{\rho_i}$, where K_i is the path length.

Another body of work looking at delay scheduling in wireline networks considers the Earliest Deadline First (EDF) schedule, where each packet has a deadline and the scheduler always picks the packet with the earliest deadline. For a single link in isolation, [25, 58] showed that, if each interval is locally schedulable, in the sense that for each interval $[s, t)$ the total amount of data injected after time s and with a deadline before time t is at most $t - s$, then EDF will schedule all the packets so that deadlines are met. For the case of networks with many links, [26] showed how to choose deadlines so as to match the $\frac{\sigma_i+K_i}{\rho_i}$ delay bound of WFQ.

One feature of both the above delay bounds is that they contain a term of the form $\frac{K_i}{\rho_i}$. Although, it is not hard to show that K_i and $\frac{\sigma_i}{\rho_i}$ are both lower bounds on delay that cannot be overcome, there is no reason why the term $\frac{K_i}{\rho_i}$ is intrinsically necessary, i.e., a flow does not have to experience delay $1/\rho_i$ at every hop. In [7], a *Coordinated Earliest Deadline First* (CEDF) protocol was presented, that had a much better

delay bound, namely a bound of the form $\tilde{O}(\frac{\sigma_i}{\rho_i} + K_i)$ (where the $\tilde{O}(\cdot)$ hides logarithmic factors). The most important aspect of this bound is that it does *not* contain a term of the form K_i/ρ_i, i.e., flow i does not need to experience delay $1/\rho_i$ at every hop.

Similar results have been obtained in the context of real-time scheduling in distributed systems. For instance, in [64] certain scheduling policies were shown to have "pipeline-like" behavior, where the mean end-to-end delay of non-acyclic flows was shown to be inversely proportional to the flow rates at a single hop along the flow's path, plus a constant delay for the remaining hops in the path (similar to $O(1/\rho_i + K_i)$). Our delay composition results provide similar bounds (delay inversely proportional to the rate on one hop plus a constant per-hop delay on future hops) for worst-case end-to-end delay of flows in a distributed system scheduled under preemptive or non-preemptive prioritized scheduling.

In general, wireless scheduling is a much more complicated problem since we are unable to schedule each link independently due to interference. Many models have been considered to characterize interference. For example, in the unit disk graph model, two transmitters can transmit only if they are more than a unit distance apart. In the primary interference model, we are given an interference graph and two links can transmit if they do not share any endpoints. In the secondary interference model, two links can transmit if neither of the end points of one link is equal or adjacent to either of the end points of another link. Other work considers a more physical model, in which a link can transmit if and only if its signal-to-noise-ratio is above a certain threshold.

The major problem addressed by most work in wireless scheduling is how to achieve *throughput optimality*. A schedule is said to be throughput-optimal, if it can service any set of flow rates that can be served by the optimal schedule. The work of Tassiulas and Ephremides [86] provides an algorithm that achieves throughput optimality regardless of the set of feasible links. This algorithm is sometimes known as the Max-Weight or Backpressure algorithm, and it operates by maintaining queues of data for each possible flow at each node. The urgency of a queue is then defined to be the size of a queue together with the size of the corresponding queue at the next hop along the flow path. The backpressure algorithm always tries to serve the set of queues that maximizes the aggregate urgency.

The main result of Tassiulas and Ephremides is that the backpressure algorithm is throughput-optimal in the following sense. If the (slightly augmented) flow rates $(1 + \varepsilon)\rho_i$ lie in the schedulability region, then the backpressure algorithm will keep the queues stable, if data is injected into flow i at rate ρ_i.

With regards to the delay performance of wireless schedulers, Jagabathula and Shah [37] show that under the primary interference model, if the traffic is injected according to a Poisson process in each flow, then the average end-to-end delay for flow i is at most $5K_i/\varepsilon$, as long as the flow rates ρ_i lie in the schedulability

region scaled down by a factor of 5. For secondary interference constraints, a similar result holds with the factor "5" replaced by Δ^2, where Δ is the maximum number of links that can interfere with any given link. They also show that arbitrary traffic can be Poissonized by running it through a scheduler that injects packets according to a Poisson process. However, this Poissonizer could potentially add another term of $1/\rho_i$ to the delay. Our results differ from [37], in that we are concerned with the worst-case delay of flows that lie close the boundary of the schedulability region.

There is also a large body of work that analyzes delays in a random wireless network with random mobility and random traffic patterns, for example [29, 24, 17, 66]. This is distinct from our work since we are concerned with a given network topology with given set of traffic flows.

In this work, we study the end-to-end delay bounds that can be obtained by combining the CEDF scheduler with a wireless link scheduling algorithm. Before we can present our results in more detail we must describe our model.

The Model

We consider the problem of minimizing the end-to-end delay experienced by data flows in a wireless data network with n nodes and N wireless links. Each flow i consists of a path p_i of K_i hops through the network. The traffic for flow i has rate ρ_i and burst parameter σ_i, i.e., if $A_i(s,t)$ is the amount of data injected into flow i during the time interval $(s,t]$ then $A_i(s,t) \leq \sigma_i + \rho_i(t-s)$.

A key concept in our analysis will be the concept of a schedulable region. We say that a binary N-vector χ is feasible, if all links for which $\chi_l = 1$ can simultaneously transmit without interfering with one another (χ_l is the l^{th} entry of χ). Our results will apply to any set of interference constraints that satisfy some regularity conditions. In particular, we assume that there is some distance δ such that two links do not interfere if their endpoints are at least a distance δ apart. For simplicity, we normalize distances so that $\delta = 1$. In addition, we assume that there is some constant γ, such that for any square of side L, the maximum number of links in the square that can transmit simultaneously is at most γL.

We assume that time is slotted. A *link schedule* consists of a sequence of feasible vectors, one for each time slot. If χ is scheduled during slot t, then each link for which $\chi_l = 1$ can transmit one packet during the slot. We define the link schedulability region to be the set of link rates that can be utilized under some scheduling algorithm, i.e., a rate vector r is schedulable, if and only if there exist non-negative numbers $\phi_1, \phi_2, \ldots$, and feasible vectors $\chi_1, \chi_2, \ldots$, such that $r \leq \sum_k \phi_k \chi_k$ (the inequality is component-wise) and $\sum_k \phi_k = 1$.

We assume that the flows in the network generate link rates that lie strictly within the schedulability region. That is, we assume that flow routes are fixed and we let F_l be the set of flows i that pass through

the link l. If we let $r_l = \sum_{f \in F_l} \rho_i$, then we assume that there is some vector r' that is schedulable, such that $r_l \leq (1-\epsilon)(r')_l$ for all l and for some constant ϵ.

Consider the packets injected into each flow. Our aim is to schedule these packets in such a way that the end-to-end delay remains low. We use the term *complete schedule* to refer to a link schedule combined with a specification of which packets cross each link in each time slot.

Results

- In Section 9.2.1, we show that by using a centralized algorithm to determine the link schedule, and CEDF to determine the packets to be transmitted along each link for every time slot, a worst-case end-to-end delay of approximately $\frac{\sigma_i}{\rho_i} + \sum_{l \in p_i} \frac{N}{r_l}$ can be achieved for every flow i. The key features of this delay bound are that each packet waits for time roughly σ_i/ρ_i to access the channel at the first hop. Thereafter it only waits for time roughly N/r_l at each subsequent link l.

- The above result uses a centralized scheme to find a feasible set of links to transmit at each time step. In Section 9.2.2, we show how to convert this into a distributed algorithm by decomposing the scenario using a set of L^2 grids for some $L = O(1/\epsilon)$.

- The above delay bounds were found using link scheduling algorithms that rely on an optimal decomposition of any link rate vector into a set of schedulable link sets. For cases where this is not feasible, we also examine the delay bounds that can be achieved if we use the Max-Weight algorithm for constructing the link schedules (but not the individual packet schedules).

- For wireline networks, if we look at each link in isolation, it is known that if the naive schedulability condition holds for each time interval taken in isolation then all the deadlines can be met (using an Earliest-Deadline-First schedule). In Section 9.2.4, we examine the wireless counterpart to that statement and show that for wireless networks it does not hold in general.

- In Section 9.2.5, we briefly describe a simulation scenario to demonstrate the benefits of our approach. We conclude in Section 9.2.6, by showing how the theoretical algorithms discussed in this section can be used to motivate more practical schemes based on random access.

9.2.1 Centralized Scheduling to Minimize End-to-End Delay

The problem of minimizing the end-to-end delay of packets in a wireless network comprises of two sub-problems. The first, called the *link schedule*, determines a feasible set of links on which to transmit packets at each time slot. In the second sub-problem, called the *packet schedule*, each link determines which packets should be transmitted at any given time slot.

We first describe the general service guarantee that any link scheduling algorithm needs to provide for each link in the network, in order to support delay guarantees. We then assume the presence of an oracle that provides a decomposition of the rate vector into a convex combination of feasible vectors, where each vector denotes a set of independent links that can transmit simultaneously within the network (this oracle could be realized via linear programming; in Section 9.2.2, we show how to realize the oracle in a distributed manner with an arbitrarily small loss in achievable rate). We present a centralized algorithm to determine the link schedule. The packet scheduling is performed in a localized and distributed manner. We subsequently derive the delay guarantee for each packet.

Objective of the Link Schedule

In order to provide delay guarantees, we need to schedule the links such that for any arbitrary interval $[s, t)$, the service received by each link does not deviate unboundedly from its average rate. If $C_l(t)$ is the service received by link l up to time t, then the service guarantee for any interval $[s, t)$, in general, can be expressed as,

$$C_l(t) - C_l(s) \geq r_l(t - s) - s_l \tag{9.23}$$

where r_l is the rate of link l. The objective of the link scheduling algorithm is then to minimize s_l, across all the links in the network. In this section, we consider a centralized link scheduling algorithm with a bounded value for s_l, and in subsequent sections discuss distributed alternatives to the link scheduling problem.

A Centralized Link Scheduling Algorithm

For an input-buffered crossbar switch, given a decomposition of the rate matrix, r, into a convex combination of permutation matrices χ_k, as $r \leq \sum_{k=1}^{\kappa} \phi_k \chi_k$, Chang et. al. [13] used the Weighted Fair Queuing algorithm to schedule the crossbar switch. This algorithm approximately ensures that the amount of time the connection pattern of the crossbar switch is set according to the permutation matrix χ_k, is proportional to ϕ_k. According to this algorithm, tokens are generated for each permutation matrix and the virtual finishing times for the l^{th} token of permutation matrix k is set as l/ϕ_k. The tokens are served in the increasing order of their virtual finishing times, setting the crossbar's connection pattern according to permutation matrix χ_k when serving a token belonging to χ_k. Based on this algorithm, minimum and maximum service guarantees are derived for each link l within the crossbar switch for any arbitrary time interval $[s, t)$. Let $C_l(t)$ denote the service received by link l up to time t. Let E_l denote the subset of $\{1, 2, \ldots, \kappa\}$ such that for every $k \in E_l$, the permutation matrix χ_k has a nonzero element corresponding to link l. The service guarantees

derived in [13] for each link l are,

$$\sum_{k \in E_l} \phi_k(t-s) - s_l \leq C_l(t) - C_l(s) \leq \sum_{k \in E_l} \phi_k(t-s) + s_l \tag{9.24}$$

where $s_l = min(\kappa, |E_l| + \sum_{k \in E_l} \phi_k(\kappa - 1))$ (by the theory of linear programming, κ is bounded by N, the number of links in the network).

This result was presented for crossbar switches, where all input ports are connected to all output ports. The only constraints on scheduling are that each port can either transmit or receive at most one packet per time slot. In contrast, we are interested in a feasible link schedule with similar guarantees on service for each link, under arbitrary schedulability constraints. Towards this end, we start by assuming an oracle that provides a decomposition of the rate vector into a feasible set of vectors, such that the links activated under each vector can be simultaneously scheduled (we show how such an oracle can be achieved using a distributed algorithm in the next section). The result in [13] holds even when the we use link rate vectors rather than permutation matrices, and hence can be applied to our wireless setting (there exists a decomposition into $\kappa \leq N$ feasible vectors).

Packet Scheduling along Each Link: Coordinated Earliest Deadline First

Given the link schedule, we next need to determine how each link schedules individual packets so as to minimize the worst-case end-to-end delay. We adopt a Coordinated Earliest Deadline First (CEDF) scheme, similar to [7]. The scheme combines randomization and coordination with EDF to guarantee an end-to-end delay bound of $O(1/\rho_i + \sum_{l \in p_i} \frac{1}{r_l})$ with high probability, under arbitrary wireless schedulability constraints.

Let r_l denote the rate at which packets can be transmitted across a link l, that is $r_l = \sum_{k \in E_l} \phi_k$. We assume the following condition on the flow rates through link l:

$$\sum_{i \in F_l} \rho_i \leq (1-\epsilon) r_l \frac{\sum_{k \in E_l} \phi_k G_l - s_l}{G_l} \tag{9.25}$$

for some $\epsilon > 0$, where G_l provides a bound on the maximum delay incurred by any packet at link l (more details follow). Note that the factor $U_l = \sum_{k \in E_l} \phi_k G_l - s_l$, quantifies the minimum amount of time link l is scheduled to transmit in any interval of length G_l. Therefore, $r_l \frac{U_l}{G_l}$ denotes the minimum effective rate of link l, achieved using the link scheduling algorithm described above. The parameter $(1-\epsilon)$ represents a link utilization factor, that plays a crucial role in allowing us to meet packet deadlines with high probability.

The Coordinated-EDF schedule works as follows. Each packet p of flow i is assigned deadlines $D_1, D_2, \ldots,$ D_{K_i}, for each link along p's path. The deadlines of packets along link l are based on the parameter G_l, which is independent of the individual rates of flows through the link. The deadline for the first hop D_1 is

defined as $rand_i + G_{l_1}$ time after p's injection into the network, where $rand_i$ is a random number chosen proportional to $1/\rho_i$. This randomness serves to spread out the deadlines on future hops so that packets don't all arrive at a node together. The deadlines of subsequent hops are set as $D_{k+1} = D_k + G_{l_{k+1}}$. CEDF chooses the packet with the earliest deadline to schedule at each time slot, and ties are broken arbitrarily. Thus, each packet suffers a delay proportional to σ_i/ρ_i at the first hop, and then suffers a smaller delay at all subsequent hops.

The random number $rand_i$ used to assign deadlines is chosen from an interval of size T_i. When T_i is as large as $2/\epsilon\rho_i$, we find that the deadlines are chosen far enough apart with high probability. We define another parameter M to denote the cycle period of the deadlines, such that once the deadlines are chosen within an interval of length M, the same deadlines can be repeated on future periods as well. We define a set of parameters as follows:

$$T_i = 2^{\lceil \log_2 \frac{2}{\epsilon\rho_i} \rceil}; \quad M = \max_i T_i; \quad S_i = T_i\rho_i(1 + \frac{\epsilon}{2}) \tag{9.26}$$

Note that such a definition of T_i's ensures that M is an integral multiple of the T_i's. Also, note that S_i/T_i is defined to be slightly larger than ρ_i.

Let N be the number of links in the network and r^* denote the maximum rate of any link. We define G_l, the amount by which deadlines are incremented for each link l as,

$$G_l = \frac{s_l + \frac{\alpha}{r^*}\log_e\left(NMr^*\epsilon\right)}{\sum_{k\in E_l}\phi_k}; \quad U_l = \frac{\alpha}{r^*}\log_e\left(NMr^*\epsilon\right) \tag{9.27}$$

where $\alpha = O(\epsilon^{-3}\log_e \frac{1}{1-p_{suc}})$, and p_{suc} is the desired success probability of the protocol (refer proof of Lemma 4).

Deadlines are chosen using tokens. For each flow i, we choose numbers $\tau_1, \tau_2, \ldots, \tau_{M/T_i}$ uniformly at random from intervals $[0, T_i), [T_i, 2T_i), \ldots, [M - T_i, M)$, respectively. A flow-$i$ token appears at each of these time instants, τ_l, $l \leq M/T_i$, which is repeated for each period of length M (a token is released at time instants $\tau_l + yM$, for $l \leq M/T_i$ and $y = 0, 1, 2, \ldots$). Each token of flow i services at most S_i packets, and each packet needs to obtain a token and consume one unit of its capacity. Note that the notion of tokens is entirely for the purposes of accounting and assigning deadlines, and the network does not have to physically support tokens. Consider a packet p of flow i, that is injected at time t_{inj}. Suppose that the flow-i packet injected immediately prior to packet p, obtained its token at time t_{prev}. Then, packet p obtains the first flow-i token after time $\tau = \max(t_{inj}, t_{prev})$ that has enough capacity to serve one packet. The deadlines of packet p are defined as $D_1 = \tau + G_{l_1}$, $D_j = D_{j-1} + G_{l_j}$. Given the deadlines of all the packets at each hop, each link chooses the packet with the earliest deadline to serve at any given time slot on which it is

scheduled to transmit.

Deriving the Delay Bound

We shall now prove that all deadlines are met with high probability using the coordinated EDF scheme, and derive the worst-case delay bound of $O(\frac{1}{\rho_i} + \sum_{l \in p_i} \frac{1}{r_l})$ for a packet of flow i. The proof is similar to the proof in [7] for end-to-end delay in wireline networks.

Consider a link l and a time interval I. Let x packets of link l have a deadline within the interval I. In this case, we say that I services x packets at link l. Recall that the link schedule is determined by a centralized oracle that ensures that whenever link l is scheduled to transmit, it suffers no interference from other simultaneous transmissions within the network.

Lemma 4. *For a link l and an interval $I = [t - G_l, t]$, where t is a potential deadline for some packet at link l, I services at most $[\sum_{k \in E_l} \phi_k G_l - s_l]$ packets at link l, with high probability.*

Proof. Let X_i denote the number of packets of flow-i that I services at link l. Tokens are placed randomly in the intervals $[0, T_i), [T_i, 2T_i)$, etc. Each token services at most S_i packets. Therefore, in expectation, any interval I of length G_l services at most $\frac{S_i}{T_i} G_l$ packets, that is, $E[X_i] = \frac{S_i}{T_i} G_l$. Adding the expected values across all flows along link l, by linearity of expectation,

$$\begin{aligned} E[\sum_{i \in F_l} X_i] &\leq \sum_{i \in F_l} \frac{S_i}{T_i} G_l \leq (1 - \frac{\epsilon}{2}) r_l (\sum_{k \in E_l} \phi_k G_l - s_l) \\ &\leq (1 - \frac{\epsilon}{2}) U_l \end{aligned}$$

A Chernoff-type argument can be used to show that $Pr[\sum_{i \in F_l} X_i \geq U_l]$ is small. In particular,

$$Pr[\sum_{i \in F_l} X_i \geq U_l] \leq e^{-\epsilon^3 (1-\epsilon) U_l / 48}$$

Due to the periodic nature of token placement, we need to only analyze a period of length M. For any link l, there exists at most M/T_i intervals $I = [t - G_l, t]$, such that t is a deadline for a flow-i packet in that time period. The total number of such intervals I can be bounded as,

$$N \sum_i \frac{M}{T_i} \leq N \sum_i \frac{M \epsilon \rho_i}{2} \leq \frac{N M r^* \epsilon}{2}$$

Recall the definitions of G_l and U_l from Equation 9.27. The probability that link l services at least U_l packets during any interval I is at most,

$$\Big(\frac{NMr^*\epsilon}{2}\Big)\Big(e^{\epsilon^3(1-\epsilon)U_l/48}\Big) \quad \leq \quad \Big(\frac{NMr^*\epsilon}{2}\Big)\Big(\frac{1}{NMr^*\epsilon}^{\alpha\epsilon^3(1-\epsilon)/48}\Big) \quad \leq \quad 1 - p_{suc}$$

for $\alpha = \frac{48}{\epsilon^3(1-\epsilon)} log_e(\frac{1}{1-p_{suc}})$. We can suitably choose p_{suc}, the success probability of the algorithm, to be close to 1. □

Lemma 5. *If Lemma 4 holds, then every packet of every flow meets all its deadlines.*

Proof. Assume the contrary. Let D be the first deadline to be missed. Let p be the packet that misses this deadline at link l. Since packet p meets all its previous deadlines, it must have arrived for transmission at link l by time $D - G_l$. As packet p misses its deadline, it must be the case that link l is busy transmitting other packets whenever it is chosen to transmit by the link schedule during the interval $[D - G_l, D - 1/r^*]$. Let p' be such a packet transmitted along link l before packet p. As the scheduling is performed according to EDF, p' must have a deadline $D' \leq D$. Further, as packet p is the first packet to miss its deadline, $D' \geq D - G_l$. Based on the construction of the link schedule, it is guaranteed that link l is scheduled to transmit for at least a duration of $\sum_{k \in E_l} \phi_k G_l - s_l$ in any interval of length G_l. Therefore, the total length of all packets that have deadlines in $[D - G_l, D]$ is at least $(\sum_{k \in E_l} \phi_k G_l - s_l)$ (one packet is served during each time slot for which a link is scheduled). This contradicts the assumption that Lemma 4 holds. □

Lemma 4 and Lemma 5 imply that every packet of each flow i reaches its destination by time at most $\tau + \sum_{j=1}^{K_i} G_{l_j}$. We now upper-bound τ as follows.

Lemma 6. *For each packet p of flow i, injected at time t_{inj}, $\tau \leq t_{inj} + \frac{\sigma_i}{\rho_i} + \frac{4}{\epsilon \rho_i}$.*

Proof. Let t_0 be the last time before t_{inj} when no flow-i packet is waiting to obtain a token. During the interval (t_0, τ) every flow i token must consume a packet injected during (t_0, t_{inj}), and each token must have consumed at least $S_i - 1$ of its capacity. Otherwise, either p would obtain a token before time τ or the interval (t_0, t_{inj}) would contain a time when no flow-i packet is waiting to be transmitted.

The total number of flow-i tokens during (t_0, t_{inj}) is at least $\frac{\tau - t_0 - T_i}{T_i}$, with each token consuming at least $S_i - 1$ of its capacity. The total number of packets of flow i injected during (t_0, t_{inj}) is at most $\sigma_i + (t_{inj} - t_0)\rho_i$.

We therefore have the bound,

$$\begin{aligned} & \frac{\tau - t_0 - T_i}{T_i}(S_i - 1) \leq \sigma_i + (t_{inj} - t_0)\rho_i \\ \Rightarrow \quad & \frac{\tau - t_0 - T_i}{\rho_i T_i}(\rho_i T_i + 1 - 1) \leq \frac{\sigma_i}{\rho_i} + (t_{inj} - t_0) \\ \Rightarrow \quad & \tau \leq t_{inj} + \frac{\sigma_i}{\rho_i} + \frac{4}{\epsilon \rho_i} \end{aligned}$$

□

Theorem 1. *With a link scheduling algorithm that guarantees a rate ρ_l for each link l and a latency in service of at most s_l, using CEDF to schedule packets along each link, the worst-case end-to-end delay of*

packets of each flow i following path p_i can be bounded as,

$$\frac{\sigma_i}{\rho_i} + \frac{4}{\epsilon\rho_i} + \sum_{l \in p_i} \frac{N + \alpha \log_e \left(NMr^*\epsilon\right)}{r_l}$$

Proof. The proof follows directly from $s_l \leq \kappa \leq N$ and Lemmas 4, 5, and 6. □

Link scheduling via the Max-Weight algorithm

The above algorithm performed the link scheduling by calculating the ϕ_k values using a linear program and then scheduling the feasible sets χ_k using [13]. We can also perform link scheduling using the Max-Weight algorithm of Tassiulas and Ephremides [86] in the following manner. Suppose that we have a token buffer q_l that is fed with tokens at rate r_l and decreased by 1 whenever link l is served. The Max-Weight algorithm will always serve the set of links S for which $\sum_{l \in S} q_l$ is maximum. The standard stability analysis of Max-Weight can also be used to show that the value of s_l for this schedule is at most $2N/\varepsilon$ (we omit the details here for reasons of space). We can therefore use the analysis of the previous section to obtain an end-to-end delay bound of,

$$\frac{\sigma_i}{\rho_i} + \frac{4}{\epsilon\rho_i} + \sum_{l \in p_i} \frac{\frac{2N}{\epsilon} + \alpha \log_e \left(NMr^*\epsilon\right)}{r_l}$$

9.2.2 Distributed Solution based on Decomposition

In this section, we examine how to construct the link schedule in a distributed manner. We recall that the maximum distance at which links can interfere is at most 1 and the maximum number of links that can be simultaneously active in any square of side L is at most γL^2 (for example, for the unit disk graph model we have $\gamma = 1/\pi$).

Our algorithm is extremely simple and is motivated by the algorithm for Maximum Independent Set in unit-disk graphs of Hunt et al. [36]. While the discussion in Section 9.2.1 works for any arbitrary interference model, in this section we assume a geometric graph model. At a high-level the algorithm works as follows. We decompose the network into a sequence of grids of squares. In each grid there is a guard band around each square to make sure that the different squares do not interfere with each other. Each grid is offset from the previous one so that every link appears within a square, for all but a constant number of the grids. We remark that many of the ideas we use have already appeared in the literature on Max Weight Independent Set calculation in wireless networks.

Grid decomposition

We divide the whole region of the network into squares of side 1. Note that any link can lie in at most 4 such squares. We now define an arrangement of large squares of side $L = 5/\epsilon$. Grid (u, v) consists of a set of large squares whose corner points can be written as $(k_1L + u, k_2L + v)$, $((k_1 + 1)L + u - 1, k_2L + v)$, $(k_1L + u, (k_2 + 1)L + v - 1)$, $((k_1 + 1)L + u - 1, (k_2 + 1)L + v - 1)$ for some integers k_1 and k_2. A link is said to belong to a grid, if both its endpoints are contained in one of the large squares that make up the grid.

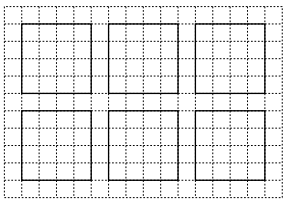

Figure 9.23: One grid (u, v).

A *grid-schedule* for grid (u, v) consists of a schedule for all the links that belong to the grid. Recall that each square can be scheduled independently, since there is no interference between neighboring squares in the grid.

Let $S^{(u,v)}(t)$ be the set of links scheduled by the schedule for grid (u, v) at time slot t. We create our schedule for the entire network by interleaving the schedules for the different grids. In particular, our complete schedule is given by,

$$
\begin{aligned}
S^{(0,0)}(0), S^{(1,1)}(0), \ldots, S^{(L-1,L-1)}(0) \\
S^{(0,1)}(0), S^{(1,2)}(0), \ldots, S^{(L-1,0)}(0) \\
\vdots \\
S^{(0,L-1)}(0), S^{(1,0)}(0), \ldots, S^{(L-1,L-2)}(0) \\
S^{(0,0)}(1), S^{(1,1)}(1), \ldots, S^{(L-1,L-1)}(1) \\
\vdots \\
S^{(0,L-1)}(1), S^{(1,0)}(1), \ldots, S^{(L-1,L-2)}(1) \\
S^{(0,0)}(2), S^{(1,1)}(2), \ldots, S^{(L-1,L-1)}(2) \\
\vdots \\
S^{(0,L-1)}(2), S^{(1,0)}(2), \ldots, S^{(L-1,L-2)}(2) \\
\vdots
\end{aligned}
$$

Lemma 7. *If the service guarantee s_l for each link l in each grid schedule $S^{(u,v)}(\cdot)$ is at most Δ, then the service guarantee in the final schedule $S(\cdot)$ is at most $L^2\Delta$.*

Proof. For each constituent schedule, we have that the amount of service given to the link in any interval of length t' is at least $r_\ell(t' - \Delta)$.

In every row of the complete schedule shown above, link l belongs to at least $L-2$ of the L grids. Consider any time interval of length t. There are $x_1 = t \mod L^2$ schedules $S^{(u,v)}(\cdot)$, that specify service in $\lfloor t/L^2 \rfloor + 1$ of these time slots. Of these schedules, at least $x_1 - 2\lceil x_1/L \rceil$ will contain the link l. In addition, there are $x_2 = L^2 - (t \mod L^2)$ schedules, that specify service in $\lfloor t/L^2 \rfloor$ time slots. Of these schedules, at least $x_2 - 2\lceil x_1/L \rceil$ will contain the link l.

Hence, the total amount of service given to link l in an interval of length t is at least,

$$\begin{aligned} & (x_1 - 2\lceil x_1/L \rceil) r_l(\lfloor t/L^2 \rfloor + 1 - \Delta) \\ & +(x_2 - 2\lceil x_2/L \rceil) r_l(\lfloor t/L^2 \rfloor - \Delta) \\ \leq\ & (((x_1 + x_2)(1 - (2/L)) - 4) r_l(\frac{t}{L^2} - \Delta) \\ =\ & (1 - \frac{2}{L} - \frac{4}{L^2}) r_l(t - L^2\Delta) \end{aligned}$$

Note that, the service rate for each link has decreased by a factor $(1 - \frac{2}{L} - \frac{4}{L^2}) \geq (1 - \frac{\epsilon}{2})$. This is not a problem, since we assumed that the flow rates scaled by a factor $1/(1-\epsilon)$ still lie in the schedulability region. Hence, we can simply scale up r_l by a factor $1/(1 - \frac{\epsilon}{2})$. □

Creating the schedule $S^{(u,v)}$

It remains to devise a schedule for each individual grid. We assume that the link rates within any square are computed in a distributed manner. Recall that, in any square of side L, the maximum number of links that can be active in any feasible schedule in any time slot is at most βL^2, for some constant β. Therefore, the total number of feasible vectors χ is at most $N^{\beta L^2}$. Note that, β, ϵ, and hence, L, are all assumed to be constant. Hence, for any square, we can in polynomial time, use linear programming to locally compute a decomposition of the rate vector for the links in the square (the decomposition is local since each link does not need to communicate outside the square). We can then use [13] to create a schedule for the square. Since the square has at most N links, the service guarantee s_l for each link can be bounded by N. Hence, by Lemma 7, the service guarantee for the entire schedule is at most L^2N and hence we can apply a similar analysis to Section 9.2.1 to show that the end-to-end delay for the entire schedule is bounded by $\frac{\sigma_i}{\rho_i} + \frac{4}{\epsilon\rho_i} + \sum_{l \in p_i} \frac{L^2 N \frac{\alpha}{r^*} \log_e \left(NMr^*\epsilon\right)}{r_l}$. Alternatively, we can use the Max-Weight algorithm to create a link

schedule for each individual square, in which case the service guarantees can be bounded by $2L^2N/\epsilon$ and the end-to-end delay can be bounded by $\frac{\sigma_i}{\rho_i} + \frac{4}{\epsilon\rho_i} + \sum_{l \in p_i} \frac{\frac{2L^2N}{\epsilon}\frac{\alpha}{r^*}\log_e\left(NMr^*\epsilon\right)}{r_l}$.

9.2.3 Improved Delay Bounds Through Randomized Link Schedules

The algorithm presented in [13] to construct the link schedule, gives us a delay bound of $O(s_l/r_l)$ for each link l. This bound, however, assumes the worst case wherein a packet for link l arrives just when its transmission slot is over, and has to wait a worst case duration before the link is scheduled for transmission again. This delay bound can be improved using randomized probabilistic techniques such as those presented in [6]. In this section, we describe how one such randomized algorithm can be adapted to execute within each square in our distributed link scheduling framework (other algorithms can likewise be applied). We assume that for each square in the network, the decomposition of the rate matrix into a set of feasible independent link transmissions for links within the square is available, that is, $R = \sum_{k=1}^{\kappa} \phi_k P_k$.

The randomized algorithm simply schedules the feasible link matrices in a random order, such that each matrix P_k appears with a rate ϕ_k. We assume that for some frame length η, we can find integers l_k, such that $\phi_k = l_k/\eta$. l_k tokens are generated for each matrix P_k, and tokens are chosen randomly from the set of all tokens. The matrices are scheduled according to the order in which tokens are chosen. It was shown in [6] that, with a probability of $1-\epsilon$, the delay in service, s_l, for a link l can be bounded as,

$$s_l \rightarrow r_l \sqrt{A(\frac{1}{r_l} - 1)\eta} \tag{9.28}$$

for $\eta \rightarrow \infty$, where $A < \frac{1+10\epsilon}{4}$. The algorithm can be executed independently for each square in the distributed link scheduling framework, with different frame lengths for each square. As each link l is served at a rate r_l within each square to which it belongs, the probability that a link l is chosen for transmission can be assumed to be uniform across all the squares to which it belongs. For a length η taken as the maximum frame length of any square on any grid within the network, the service latency s_l can be bounded by Equation 9.28 above. For grid selections to which a link does not belong, the rate achieved for the link is zero. As $\eta \rightarrow \infty$, this delay can be assumed to be negligible, as there exist only 2 grids where the link is not included in the schedule for every η grids. The delay bound for this randomized algorithm tends to be significantly better than the bound provided in [13], due to the presence of the square root.

Let $S^{(r,s)}(t)$ denote the set of links scheduled under grid (r,s) at time slot t. Let $\eta^{(r,s)}$ denote the LCM of the frame lengths for the schedules at each of the squares under grid (r,s). Let Π denote a random permutation of the set $\{0,1,2,\ldots,\eta^{(r,s)}\}$ (the random permutation can be different for each square within each grid (r,s), but for simplicity of notation, we drop the superscripts for Π). The complete schedule under

the random permutation scheme is then given by,

$$
\begin{aligned}
&S^{(0,0)}(\Pi(0)), S^{(1,1)}(\Pi(0)), \ldots, S^{(L-1,L-1)}(\Pi(0)) \\
&S^{(0,1)}(\Pi(0)), S^{(1,2)}(\Pi(0)), \ldots, S^{(L-1,0)}(\Pi(0)) \\
&S^{(0,2)}(\Pi(0)), S^{(1,3)}(\Pi(0)), \ldots, S^{(L-1,1)}(\Pi(0)) \\
&\vdots \\
&S^{(0,L-1)}(\Pi(0)), S^{(1,0)}(\Pi(0)), \ldots, S^{(L-1,L-2)}(\Pi(0)) \\
&S^{(0,0)}(\Pi(1)), S^{(1,1)}(\Pi(1)), \ldots, S^{(L-1,L-1)}(\Pi(1)) \\
&S^{(0,1)}(\Pi(1)), S^{(1,2)}(\Pi(1)), \ldots, S^{(L-1,0)}(\Pi(1)) \\
&S^{(0,2)}(\Pi(1)), S^{(1,3)}(\Pi(1)), \ldots, S^{(L-1,1)}(\Pi(1)) \\
&\vdots \\
&S^{(0,L-1)}(\Pi(1)), S^{(1,0)}(\Pi(1)), \ldots, S^{(L-1,L-2)}(\Pi(1)) \\
&S^{(0,0)}(\Pi(2)), S^{(1,1)}(\Pi(2)), \ldots, S^{(L-1,L-1)}(\Pi(2)) \\
&S^{(0,1)}(\Pi(2)), S^{(1,2)}(\Pi(2)), \ldots, S^{(L-1,0)}(\Pi(2)) \\
&S^{(0,2)}(\Pi(2)), S^{(1,3)}(\Pi(2)), \ldots, S^{(L-1,1)}(\Pi(2)) \\
&\vdots \\
&S^{(0,L-1)}(\Pi(2)), S^{(1,0)}(\Pi(2)), \ldots, S^{(L-1,L-2)}(\Pi(2)) \\
&\vdots
\end{aligned}
$$

Note that each link needs to know this random permutation apriori to know when it is scheduled to transmit. With the above randomized link scheduling algorithm and the coordinated earliest deadline first algorithm to schedule packets at each link, we can bound the worst-case end-to-end delay of packets of flows through the following corollary of Theorem 1.

Corollary 3. *Using a random permutation of the link schedules within each square, and using the coordinated earliest deadline first algorithm to schedule the packets at each link, packets of a flow i following a path p_i has with high probability a worst-case end-to-end delay of*

$$\frac{\sigma_i}{\rho_i} + \frac{4}{\epsilon \rho_i} + \sum_{l \in p_i} \frac{r_l \sqrt{A(\frac{1}{r_l} - 1)\eta} + \frac{\alpha}{r^*} \log_e \left(NMr^*\epsilon\right)}{r_l}$$

A is the positive solution of $\sum_{k \geq 1} (4k^2 A - 1) e^{-2k^2 A} = \epsilon$.

Likewise, the random-phase periodic competition scheduler [6] can be used to schedule the links within each square, wherein tokens are generated for each matrix with period $1/\phi_k$ and with a random phase shift of V_k/ϕ_k, where V_k is a random number between 0 and 1. Here again, a bound of $s_l \rightarrow |E_l| + r_l(2 + \sqrt{2\kappa ln(2\eta+1)})$ can be obtained for η denoting the maximum frame length of any square within the network. The non-frame-based schedulers from [6] can also be adapted to apply to each of the individual squares.

9.2.4 Local vs. Global Schedulability in Wireless Networks

In this section, we present an example to demonstrate that in wireless networks (unlike wireline networks), local schedulability of deadlines in every neighborhood does not necessarily guarantee that there exists a feasible global schedule of all the packets. We show an example, wherein for every sub-interval, there exists a feasible schedule that ensures that the packets whose arrival time and deadline lie within the sub-interval, all meet their respective deadlines. Likewise, for any subset of the nodes, there exists a schedule such that packets originating at those nodes whose arrival time and deadline lie within the interval, all meet their respective deadlines. In addition, there exists a global schedule for the entire interval that ensures that all packets are transmitted within the interval (not necessarily meeting all packet deadlines). Yet, we show that there does not exist a global schedule that meets all the deadlines of packets. Further, we show that the tardiness of packets (the minimum amount of time by which some packet in the network is delayed beyond it deadline), grows as a function of the number of packets each node has to transmit.

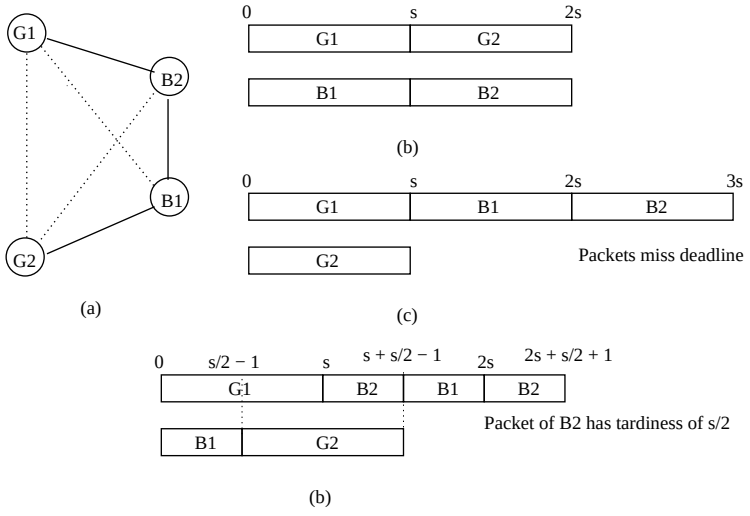

Figure 9.24: Figure illustrating the example

We consider a 4 node network of transmitters, as shown in Figure 9.24 (for concreteness, we can assume that each transmitter is transmitting to a receiver that is close by). The interference relationships between transmitters are also shown. A solid edge between two nodes denotes that the two nodes interfere with each

other and cannot simultaneously transmit in the same time slot. A dashed edge between two nodes denotes that the two nodes may simultaneously transmit. Each node has s packets that it needs to transmit. All packets arrive at time zero. Packets from nodes G_1 and G_2 have a deadline of s time slots, and those at B_1 and B_2 have a deadline of $2s$ time slots. A node can transmit at most one packet during any given time slot.

Observe that, all packets within the network can be scheduled within $2s$ time slots, if deadlines of packets are disregarded. This is achieved by first transmitting packets of G_1 and B_1 simultaneously during the first s slots, and then transmitting packets of G_2 and B_2 during the next s time slots. This is shown in Figure 9.24(b). Next, we show that for each sub-interval, all packets whose arrival and deadline lie within the sub-interval can be scheduled within that sub-interval. Packets within the interval $[0, s]$ (all smaller intervals do not contain any packets), namely packets from G_1 and G_2 can be simultaneously transmitted. Other intervals of size larger than s starting at time zero, contain the same set of packets, and are therefore schedulable in s slots.

Next, consider any subset of nodes in the system. For the subset $\{G_1, G_2, B_1\}$, G_1 and G_2 can transmit during the first s slots and B_1 can transmit during the next s slots (similarly for the subset $\{G_1, G_2, B_2\}$). For the subset $\{G_1, B_1, B_2\}$, G_1 and B_1 can be transmitted during the first s slots and B_2 can transmit during the next s slots (a similar schedule exists for the subset $\{G_2, B_1, B_2\}$). Schedules for all smaller subsets of nodes can be constructed from these schedules.

Finally, let us consider the entire interval $[0, 2s]$. Nodes G_1 and G_2 need to transmit their s packets in the first s slots in order to meet their deadlines. However, B_1 and B_2 cannot simultaneously transmit, and therefore require $2s$ time slots to transmit all their packets, implying that s packets will miss their deadline at time $2s$. This is shown in Figure 9.24(c).

Further, we show that the tardiness of packets grows as a function of s.

Lemma 8. *For the above example, tardiness smaller than $\lceil \frac{s}{2} \rceil$ for every packet cannot be achieved.*

Proof. To prove this, let us assume the contrary. Suppose there exists a schedule where all the packets have a tardiness of at most $\lceil \frac{s}{2} \rceil - 1$ time slots. Packets of G_1 and G_2 have at most this tardiness, so they need to be scheduled by time $s + \lceil \frac{s}{2} \rceil - 1$. Therefore, on at least $\lfloor \frac{s}{2} \rfloor + 1$ time slots, G_1 and G_2 need to transmit simultaneously. Packets of B_1 and B_2 can accompany packets of G_1 and G_2, whenever G_1 and G_2 are not transmitting simultaneously. After such transmissions from B_1 and B_2, there are at least $\lfloor \frac{s}{2} \rfloor + 1$ packets of each of B_1 and B_2 still to be transmitted at time $s + \lceil \frac{s}{2} \rceil - 1$. Therefore, the time taken to transmit all packets is at least $s + \lceil \frac{s}{2} \rceil - 1 + 2(\lfloor \frac{s}{2} \rfloor + 1) = 2s + \lfloor \frac{s}{2} \rfloor + 1$. As $\lfloor \frac{s}{2} \rfloor + 1 > \lceil \frac{s}{2} \rceil - 1$, some packet has a tardiness of at least $\lceil \frac{s}{2} \rceil$, yielding a contradiction. □

The above lemma implies that the tardiness of packets can be made to grow with the number of packets each node has to transmit. Alternatively, if each node has at most one packet to transmit, then the above example can be transformed into an example where the tardiness grows with the number of nodes in the network. Given the graph G above, one can construct the graph G' as follows. Each node with s packets in G, can be thought of as clique of s nodes (each node interfering with every other node in the clique) each having only one packet to transmit. A link between two nodes in G is replaced with links between every pair of nodes in the two cliques corresponding to the two nodes in G'. In G' as in G, the tardiness grows as a function of s.

9.2.5 Evaluation

We conduct our experiments on two simple network scenarios, each with one long flow whose end-to-end packet delay is measured, and several short interfering flows. For each scenario, the link schedule is constructed using either Chang's algorithm [13] or using the Max-Weight algorithm. Both these algorithms are approximate, as in our implementation we only consider a fixed number of independent sets of links on which to transmit at each time slot, and do not consider all possible independent sets of links. For the packet schedule, the coordinated EDF scheme is compared against Weighted Fair Queuing at each link. We use a simplified version of the CEDF scheme, similar to [7]. The deadline for the first hop is chosen from the interval $[t_{inj}, t_{inj} + \frac{1}{\rho_i}]$, where t_{inj} is the time when the packet is injected. For every future hop, the deadline is set as one packet service time more than the deadline at the previous hop. Each link serves the packet with the earliest deadline at each time step.

The link speed is assumed to be 1 Mb/s and all packets are of size 1000 bits. Therefore, each link takes 1ms to service a packet. Buffers are large enough that no packet is dropped in any of the experiments. In our interference model, two links are said to interfere, if either of the endpoints on one link is equal or adjacent to either of the end points of the other link. We study the average end-to-end delay of packets (service time and queuing time at each link along its path) belonging to the long session. The plots for 98-percentile end-to-end delay were similar and are not shown here due to space constraints.

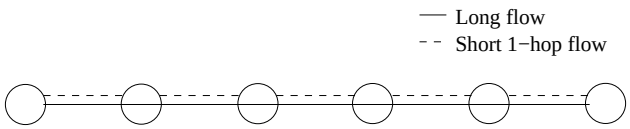

Figure 9.25: Network 1

The first network we consider is shown in Figure 9.25. It consists of one long flow through the longest path in the network, and short single-hop flows along each link. We vary the rate ρ_0 of the long flow, and

choose the rate of each of the short flows as $0.3 - \rho_0$. We plot the mean end-to-end delay of the packets of the long flow as a function of the number of hops on its path, for different values of ρ_0. Figure 9.26 shows the plot for WFQ and Figure 9.27 shows the plot for CEDF.

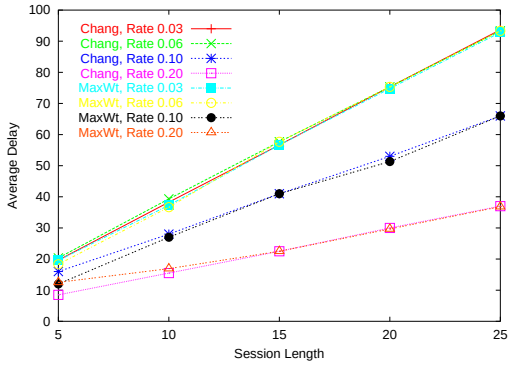

Figure 9.26: Average delay of long session under WFQ for network 1

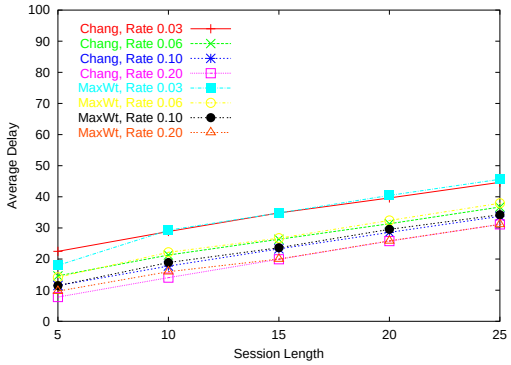

Figure 9.27: Average delay of long session under CEDF for network 1

We observe that the delay of the long flow under CEDF is in accordance with and has the same form as our analytical results. The delay increases as $O(1/\rho_0 + K_0)$, where K_0 is the number of hops of the long flow (the slope of the curves are independent of the flow rate). In contrast, for WFQ, the delay is observed to increase as $O(K_0/\rho_0)$ (the slope of the curves increases with decreasing flow rate). Further, the performance of the Max-Weight algorithm is observed to be similar to that of the Chang algorithm.

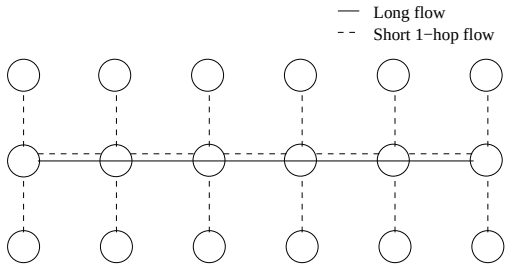

Figure 9.28: Network 2

The second network we consider is shown in Figure 9.28. This network has additional interference links, two from each node, with short 1-hop sessions along each link. The rate ρ_0 of the long flow is varied, and the rate of each of the 1-hop flows is assumed as $0.15 - \rho_0$. Figure 9.29 and 9.30 plot the mean end-to-end delay of the long flow as a function of the flow length and rate, for WFQ and CEDF, respectively.

The results are more pronounced under this network scenario. WFQ incurs significantly larger delay compared to CEDF for larger network sizes. Also, Max-Weight performs better than the Chang algorithm for link scheduling, for both packet scheduling algorithms.

We also implemented Max-Weight to perform both the link and packet scheduling, by considering a separate queue for each flow along every link. For both network scenarios, the delay for the long flow (not

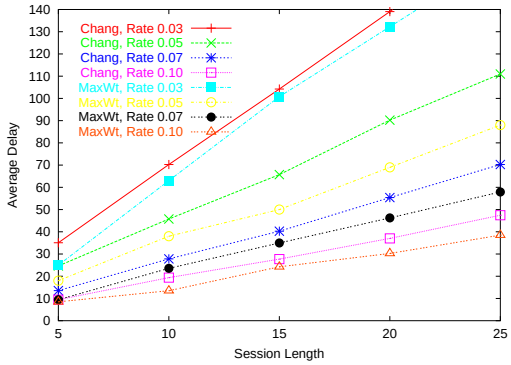

Figure 9.29: Average delay of long session under WFQ for network 2

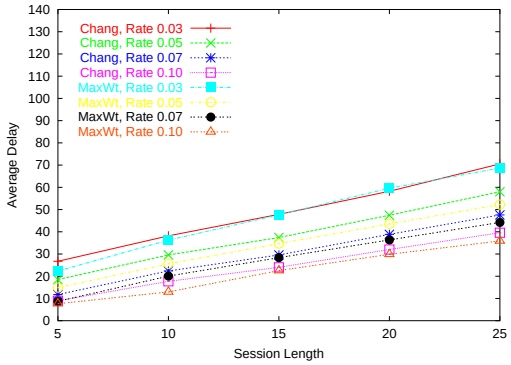

Figure 9.30: Average delay of long session under CEDF for network 2

shown here due to space constraints) was observed to be marginally poorer than the corresponding values obtained when using the Max-Weight algorithm for the link schedule and WFQ for the packet schedule.

9.2.6 Implementation Issues

In this work, our primary goal has been to determine what delay bounds are theoretically possible in the wireless setting. We believe that this type of study has value, since it indicates the type of delay performance that one should aim for when designing practical network protocols for delay-constrained traffic. However, some of our proposed techniques, such as the grid decomposition for the distributed algorithm may be difficult to implement in hardware. Hence, in this section, we briefly describe ways in which we could implement a scheduler with a small delay bound in a manner that is consistent with practical schedulers for mobile ad-hoc networks.

We first remark that the practicality of the proposed approach is only really an issue for the link scheduling algorithm. The packet scheduling component the CEDF algorithm can be implemented extremely simply in a distributed manner. We simply have to assign a delay at the first hop of the flow to create an initial deadline and thereafter add a locally computed amount to the deadline at each hop.

As already discussed, for link scheduling it may be difficult to reliably carry out the grid decomposition and the optimal feasible scheduling in a distributed manner. Hence we propose that the Max-Weight algorithm based on token buffers q_l be used for the link scheduling component, since recent work have shown ways in which the Max-Weight algorithm can be approximated using a distributed algorithm. Moreover, Section 9.2.5 indicates that the performance of Max-Weight is similar to that of the approach based on Chang et al.

We now briefly outline some of the methods that have been proposed for practical implementation of the Max-Weight algorithm in ad-hoc networks. In [20], Eryilmaz et al. provide a fully distributed implementation of a *pick-and-compare* approximation to Max-Weight. Pick-and-compare algorithms were introduced by

Tassiulas [85] and operate by continually picking a random feasible set and then only switching from the current feasible set to the new feasible set if the aggregate weight of the new set is more than the current set. It is known that as long as there is a non-zero probability of picking the Max-Weight set then such algorithms retain the throughput-optimality property. In [20], Eryilmaz et al. show how both the "pick" and "compare" phases of the algorithm can be realized by a fully distributed gossiping protocol.

Such a technique could also be used in our context for finding the Max-Weight subset. However, we remark that the delay performance is unlikely to be very good, since it could take a long time for the "pick" phase to find a new feasible set that is better than the old one and so the induced link delay for this protocol could be large during the times that a link is not in the current feasible set.

The remaining approaches are all variants of the random access mechanism used by the 802.11 protocol. In [31], Gupta et al. propose a random access scheme, which in our setup would cause each link to access the channel with probability $f(q_l)/W$, where $f(\cdot)$ is an increasing function and W is a parameter that increases/decreases depending on whether or not previous transmissions have been successful. The fact that the access probability depends on q_l ensures that links with a large buffer are more likely to transmit, and so the scheme approximates Max-Weight. In [31] it is shown that, if W is updated correctly then the access probabilities converge to the "correct" values for the current level of network congestion.

The next two protocols work directly with the 802.11 backoff mechanism, where each node[1] has a backoff counter, which counts down whenever the node senses that the channel is idle. When the counter hits zero then the node transmits. If the transmission is successful, then the counter is reset to a random amount between 1 and some fixed parameter $cw_{\min}$. If the previous transmission was not successful, then the range for the subsequent counter selection is doubled in size from the range that was previously used.

The standard 802.11 implementation does not have any mechanism for adapting the procedure according to any measure of urgency such as q_l. However, recent work has demonstrated that simple changes can make the protocol reflect urgency in an effective manner. For example, [4] Akyol et al. looked at ways to implement backpressure protocols in the 802.11 framework and proposed a scheme, in which the value of $cw_{\min}$ for a node was reduced whenever the node determined that its urgency weight was larger than the urgency weights of nodes in its immediate neighborhood. In [92], Warrier et al, considered an alternative approach in which $cw_{\min}$ was reduced whenever the urgency was larger than a fixed threshold.

[1]The standard 802.11 protocol has a separate counter for each node. However, it is easy to adapt this to the case where each node has a separate counter for each of its adjacent links.

Chapter 10

Conclusion and Future Work

In this thesis, we have developed a new reduction-based methodology for analyzing the end-to-end delay and schedulability of real-time jobs in distributed systems. We have derived a simple delay composition rule, that determines the end-to-end delay of a job in terms of the computation times of all other jobs that execute together with it. Having derived the delay composition theorem for pipelined distributed systems, we have extended it to Directed Acyclic Graphs and non-acyclic graphs as well, under both preemptive and non-preemptive scheduling. The result makes no assumptions on periodicity and is valid for periodic and aperiodic jobs. It applies to fixed and dynamic priority scheduling, as long as all jobs have the same relative priority on all stages on which they execute. The delay composition result enables a simple reduction of the distributed system to an equivalent hypothetical uniprocessor that can be analyzed using traditional uniprocessor schedulability analysis to infer the schedulability of the distributed system. Such a reduction significantly reduces the complexity of analysis and ensures that the analysis does not become exceedingly pessimistic with system scale, unlike existing analysis techniques for distributed systems such as holistic analysis and network calculus.

We developed an algebra based on the reduction-based analysis methodology. The operands of the algebra represent workloads on composed subsystems, and the operators such as PIPE and SPLIT define ways in which subsystems can be composed together. By repeatedly applying the operators on the operands representing resource stages, any distributed system can be systematically reduced to an equivalent uniprocessor that can be analyzed to determine end-to-end delay and schedulability properties of all jobs in the original distributed system.

While the above reduction-based techniques reduce the distributed system to an equivalent uniprocessor, it suffers from pessimism that arises due to the mismatch in the constraints of the distributed workload and the assumptions made by the uniprocessor task model, for the case of periodic tasks. To overcome this problem, we developed a new uniprocessor system model with mode changes, which we call, flow-based mode changes, motivated by the novel constraints of distributed workload transformation. In this model, transition of a job from one resource to another in the distributed system, is modeled as mode changes on

the uniprocessor. We presented a new iterative solution to compute the worst-case end-to-end delay of a job in the new uniprocessor task model. Reducing the distributed system to a uniprocessor with mode changes, enables much tighter schedulability analysis as demonstrated by our simulation studies.

We presented a new concept of structural robustness, which refers to the robustness of the end-to-end timing behavior of tasks in a distributed system towards unexpected timing violations in individual execution stages. We quantitatively defined the structural robustness metric with respect to the individual execution times of tasks on resources. We showed how the structural robustness of an execution graph can be improved by efficiently allocating resources to individual execution stages, thereby reducing the sensitivity of the worst-case end-to-end delays of tasks to unpredictable timing violations. Evaluation showed that our algorithm was able to reduce the number of deadline misses due to unpredictable violations in the worst-case execution times of tasks on individual stages by 40-60%. This approach will be extremely important for soft real-time systems with timing uncertainties and systems where worst-case timing is not entirely verified. We hope that future work will apply the concept of structural robustness to other systems outside the scope of the model assumed in this work.

The theory developed in this thesis was adapted to the context of wireless networks. We developed a bandwidth allocation scheme for elastic real-time flows in multi-hop wireless networks. The problem is cast as one of utility maximization, where each flow has a utility that is a concave function of its flow rate, subject to delay constraints. A flow obtains no utility if its delay constraints are violated. The delay constraints are obtained from our end-to-end delay bounds and adapted to only use localized information available within the neighborhood of each node. A constrained network utility maximization problem is formulated and solved, the solution to which results in a distributed algorithm that each node can independently execute to maximize global utility.

We also extended the end-to-end delay results obtained for distributed systems to the context of multi-hop wireless networks in the presence of arbitrary schedulability constraints. We considered the problem of minimizing end-to-end worst-case delay bounds in wireless networks and showed that by using a Coordinated EDF strategy we could ensure that a packet from flow i only needs to experience a delay of roughly $\frac{\sigma_i}{\rho_i}$ at its initial hop. Thereafter, it only needs to experience delays at its subsequent hops for which the dominant factor is $\frac{N}{r_l}$ (independent of the flow's rate, similar to our delay composition results). The extent to which worst-case end-to-end delay can be minimized in wireless networks under arbitrary schedulability constraints still remains an open problem.

We hope that the results developed in this thesis will aid in the development of a general theory for the analysis of delay in distributed systems. While there has been a lot of work on studying scheduling policies

for uniprocessor and multiprocessor systems, little is known with regard to which scheduling policies work well for distributed real-time systems. Our delay composition results provide insights into when preemptive scheduling performs better than non-preemptive scheduling and vice-versa. We need to gather a much more comprehensive understanding of various classes of scheduling policies including preemptive versus non-preemptive scheduling, fixed versus dynamic priority scheduling, and prioritized versus partitioned (each job has a reserved partition during which it executes) scheduling. We believe that the theory we have developed so far, presents the groundwork towards making crucial breakthroughs towards solving this problem.

In our study, we have predominantly considered only work-conserving scheduling policies. While non-work conserving policies tend to increase the delay incurred by a task, they are nonetheless important from the perspective of system safety and in the ability to verify that the system will always execute within states that are deemed safe. Thus, it is also important to study a mix of work conserving and non-work conserving scheduling policies.

In many distributed systems, especially in server farms and in networks, jobs could potentially traverse one of several valid routes through the system. The routing policy determines the sequence of resources through which the job is routed, and presents an additional level of complexity that was not present in uniprocessor or multiprocessor systems. The theory can be extended to optimize the routes followed by jobs. Further, in this thesis we assume that jobs require only a single resource at any given time (although they may need different resources at different times). We do not consider jobs that simultaneously require two or more resources. In order to handle systems that have tasks that require two or more resources simultaneously, we hope to develop an *AND primitive* as part of the algebra. Many different semantics are possible for simultaneous resource consumption. Semaphores and blocked execution is a well studied model outside the realm of real-time computing. Alternatively, it is possible that the scheduling at both resources are preemptive in nature (e.g., a Graphics Processing Unit and a generic processor, both scheduled preemptively), or that one resource is preemptively scheduled, while the other is scheduled in a non-preemptive manner. We currently still do not have the insights to develop analysis techniques that can handle such generic task and resource models efficiently. Nevertheless, this is certainly a very important, interesting, and challenging problem that the research community needs to address in the future.

The analysis methodology developed in this work applies a test for the schedulability of each task in the system. Thus, the test needs to be repeated separately for every task in the system, in order to determine that the entire system is schedulable. An alternative methodology is to obtain a single test that determines the schedulability of all tasks in the system. Utilization bounds are an example of such analyses, wherein a single bound, if satisfied, guarantees that all the tasks in the system are schedulable. While such utilization

bounds tend to be more pessimistic than per-task tests, they are easier to apply and are better suited to quickly determine the schedulability of large systems. More efficient per-task tests can later be conducted in order to obtain tighter bounds, or deeper insights into the functioning of the system, and to determine potential guidelines as to how the system design can be improved.

It would also be interesting to combine the advantages of delay composition theory with other generic analysis techniques such as network calculus. Delay composition theory tends to be much less pessimistic than network calculus, but is not as general. For instance, network calculus admits any scheduling policy to be used at each node in the distributed system. Combining the tightness of the delay composition results with a network calculus-type system model to leverage its generality, would significantly improve the scope of applicability of the theory.

The philosophy of compositional reduction-based analysis can be extended to other end-to-end properties such as throughput, stability, robustness, security, and functional correctness as well. If feasible, this can provide a much deeper understanding of these properties, while providing efficient and accurate ways of analyzing them.

Finally, the applicability of the theory can be extended to areas outside computing as well, such as project management. Any industrial project involves several jobs, each of which need to be completed within time constraints and involve processing by a sequence of resources. Each resource typically involves a combination of people, machines, and raw materials that need to be available simultaneously. Problems of admission control, resource provisioning, performance optimization, robustness, and cost minimization are typical in project management, just as in any distributed system. Some of the theory that we have developed so far and the problems that we still face apply in the context of project management as well. The reduction-based theory that we have developed would be immensely valuable in providing crucial insights and simple analyses during the design and management of large and complex projects. We envision that this cross-cutting research between computing and management will impact the way large organizations, irrespective of their business discipline, deal with their projects.

In the financial sector, quantitative analysis is a field that functions at the intersection of finance and distributed system computing. Thousands of bytes of data streams need to be processed by a sequence of computing units, each performing very specialized measurement, estimation, and prediction tasks, as in a distributed system. The data streams are extremely time sensitive with millions of dollars at stake. These are high-end, computationally intensive and time critical real-time applications that can greatly benefit from theoretical insights and design principles such as those initiated by this thesis.

References

[1] IEEE 802.11 WG, Draft Supplement to Part II: Wireless LAN Medium Access Control and Physical Layer Specifications: Medium Access Control Enhancements for Quality of Service (QoS).

[2] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: The multiple-node case. *IEEE/ACM Transactions on Networking*, 2(2):137 – 150, 1994.

[3] T. Abdelzaher and C. Lu. A utilization bound for aperiodic tasks and priority driven scheduling. *IEEE Transactions on Computers*, 53(3), March 2004.

[4] U. Akyol, M. Andrews, P. Gupta, J. Hobby, I. Saniee, and A. Stolyar. Joint scheduling and congestion control in mobile ad-hoc networks. In *INFOCOM*, pages 619–627, 2008.

[5] M. Andrews. Instability of FIFO in the permanent sessions model at arbitrarily small network loads. *ACM Trans. Algorithms*, 5(3):1–29, 2009.

[6] M. Andrews and M. Vojnovic. Scheduling reserved traffic in input-queued switches: New delay bounds via probabilistic techniques. *IEEE Journal on Selected Areas in Communications*, 21(4):595–605, May 2003.

[7] M. Andrews and L. Zhang. Minimizing end-to-end delay in high-speed networks with a simple coordinated schedule. In *IEEE INFOCOM*, volume 1, pages 380–388, March 1999.

[8] A. N. Audsley, A. Burns, M. Richardson, and K. Tindell. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering*, 8(5):284–292, 1993.

[9] R. Bettati and J. W. Liu. Algorithms for end-to-end scheduling to meet deadlines. In *IEEE Symposium on Parallel and Distributed Processing*, pages 62–67, December 1990.

[10] E. Bini, G. Buttazzo, and G. Buttazzo. A hyperbolic bound for the rate monotonic algorithm. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 59–66, June 2001.

[11] E. Bini, M. D. Natale, and G. Buttazzo. Sensitivity analysis for fixed-priority real-time systems. In *Euromicro Conference on Real-Time Systems*, pages 10–22, 0-0 2006.

[12] M. Caccamo, G. Buttazzo, and L. Sha. Handling execution overruns in hard real-time control systems. *IEEE Transactions on Computers*, 51(7):835–849, July 2002.

[13] C.-S. Chang, W.-J. Chen, and H.-Y. Huang. On service guarantees for input-buffered crossbar switches: A capacity decomposition approach by birkhoff and von neumann. In *International Workshop on Quality of Service (IWQoS)*, pages 79–86, June 1999.

[14] S. Chatterjee and J. Strosnider. Distributed pipeline scheduling: End-to-end analysis of heterogeneous multi-resource real-time sytems. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 204–211, May 1995.

[15] M. Chiang. To layer or not to layer: Balancing transport and physical layers in wireless multihop networks. In *IEEE Infocom*, volume 4, pages 2525–2536, March 2004.

[16] M. Chiang, S. Low, A. Calderbank, and J. Doyle. Layering as optimization decomposition: A mathematical theory of network architectures. *Proceedings of the IEEE*, 95(1):255–312, January 2007.

[17] C. Comaniciu and V. Poor. On the capacity of mobile ad hoc networks with delay constraints. *IEEE Transactions on Wireless Communications*, 5(8):2061–2071, 2006.

[18] R. Cruz. A calculus for network delay, part i: Network elements in isolation. *IEEE Transactions on Information Theory*, 37(1):114–131, January 1991.

[19] R. Cruz. A calculus for network delay, part ii: Network analysis. *IEEE Transactions on Information Theory*, 37(1):132–141, January 1991.

[20] A. Eryilmaz, A. Ozdaglar, D. Shah, and E. Modiano. Distributed cross-layer algorithms for the optimal control of multi-hop wireless networks. Submitted.

[21] N. Figueira and J. Pasquale. An upper bound delay for the virtual-clock service discipline. *IEEE/ACM Transactions on Networking*, 3(4):399–408, August 1995.

[22] V. Firoiu and D. Towsley. Call admission and resource reservation for multicast sessions. In *IEEE Infocom*, volume 1, pages 94–101, March 1996.

[23] G. Fohler and K. Ramamritham. Static scheduling of pipelined periodic tasks in distributed real-time systems. In *Euromicro Workshop on Real-Time Systems*, pages 128–135, June 1997.

[24] A. Gamal, J. Mammen, B. Prabhakar, and D. Shah. Throughput-delay trade-off in wireless networks. In *INFOCOM*, 2004.

[25] L. Georgiadis, R. Guérin, and A. Parekh. Optimal multiplexing on a single link: delay and buffer requirements. *IEEE Transactions on Information Theory*, 43(5):1518 – 1535, September 1997.

[26] L. Georgiadis, R. Guérin, V. Peris, and K. Sivarajan. Efficient network QoS provisioning based on per node traffic shaping. In *Proceedings of IEEE INFOCOM '96*, pages 102 – 110, 1996.

[27] A. Ghosal, H. Zeng, M. D. Natale, and Y. Ben-Haim. Computing robustness of flexray schedules to uncertainties in design parameters. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 550 –555, 8-12 2010.

[28] K. Gopalan, T. Chiueh, and Y.-J. Lin. Delay budget partitioning to maximize network resource usage efficiency. In *IEEE Infocom*, volume 3, pages 2060–2071, March 2004.

[29] M. Grossglauser and D. Tse. Mobility increases the capacity of ad-hoc wireless networks. *IEEE/ACM Transactions on Networking*, 10(4):477 – 486, August 2002.

[30] P. Gupta, Y. Sankarasubramaniam, and A. Stolyar. Random-access scheduling with service differentiation in wireless networks. In *IEEE Infocom*, volume 3, pages 1815–1825, March 2005.

[31] P. Gupta, Y. Sankarasubramaniam, and A. Stolyar. Random-access scheduling with service differentiation in wireless networks. In *INFOCOM*, pages 1815–1825, 2005.

[32] A. Hamann, R. Racu, and R. Ernst. A formal approach to robustness maximization of complex heterogeneous embedded systems. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 40–45, 22-25 2006.

[33] A. Hamann, R. Racu, and R. Ernst. Multi-dimensional robustness optimization in heterogeneous distributed embedded systems. In *IEEE RTAS*, pages 269–280, 3-6 2007.

[34] W. Hawkins and T. Abdelzaher. Towards feasible region calculus: An end-to-end schedulability analysis of real-time multistage execution. In *IEEE Real-Time Systems Symposium (RTSS)*, December 2005.

[35] J. Huang, R. Berry, and M. Honig. Distributed interference compensation for wireless networks. *IEEE Journal on Selected Areas in Communications (JSAC)*, 24(5):1074–1084, May 2006.

[36] H. Hunt, M. Marathe, V. Radhakrishnan, S. Ravi, D. Rosenkrantz, and R. Stearns. NC-approximation schemes for NP- and PSPACE-hard problems for geometric graphs. *Journal of Algorithms*, 26(2):238–274, 1998.

[37] S. Jagabathula and D. Shah. Optimal delay scheduling in networks with arbitrary constraints. In *SIGMETRICS*, pages 395–406, 2008.

[38] P. Jayachandran and T. Abdelzaher. A delay composition theorem for real-time pipelines. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 29–38, July 2007.

[39] P. Jayachandran and T. Abdelzaher. Delay composition algebra: A reduction-based schedulability algebra for distributed real-time systems. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 259–269, December 2008.

[40] P. Jayachandran and T. Abdelzaher. Delay composition in preemptive and non-preemptive real-time pipelines. *Invited to Real-Time Systems Journal: Special Issue on ECRTS'07*, 40(3):290–320, December 2008.

[41] P. Jayachandran and T. Abdelzaher. A delay composition theorem for real-time pipelines. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 849–857, June 2008.

[42] P. Jayachandran and T. Abdelzaher. Transforming acyclic distributed systems into equivalent uniprocessors under preemptive and non-preemptive scheduling. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 233–242, July 2008.

[43] P. Jayachandran and T. Abdelzaher. End-to-end delay analysis of distributed systems with cycles in the task graph. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 13–22, July 2009.

[44] P. Jayachandran and T. Abdelzaher. Flow-based mode changes: Towards virtual uniprocessor models for efficient reduction-based schedulability analysis of distributed systems. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 281–290, December 2009.

[45] P. Jayachandran and T. Abdelzaher. Minimizing end-to-end delay in wireless networks using a coordinated edf schedule. In *IEEE Infocom*, March 2010.

[46] P. Jayachandran and T. Abdelzaher. On structural robustness of distributed real-time systems towards uncertainties in service times. In *Submitted to Real-Time Systems Symposium (RTSS)*, December 2010.

[47] P. Jayachandran and T. Abdelzaher. Reduction-based schedulability analysis of distributed systems with cycles in the task graph. *Invited to Real-Time Systems Journal: Special Issue on ECRTS'09 (to appear)*, 2010.

[48] K. Jeffay, D. F. Stanat, and C. U. Martel. On non-preemptive scheduling of periodic and sporadic tasks. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 129–139, San Antonio, TX, 1991.

[49] B. Jonsson, S. Perathoner, L. Thiele, and W. Yi. Cyclic dependencies in modular performance analysis. In *ACM International Conference on Embedded Software (EMSOFT)*, pages 179–188, October 2008.

[50] B. Kao and H. Garcia-Molina. Deadline assignment in a distributed soft real-time system. *IEEE Transactions on Parallel and Distributed Systems*, 8(12):1268–1274, 1997.

[51] F. P. Kelly, A. K. Maulloo, and D. K. H. Tan. Rate control for communication networks: Shadow prices, proportional fairness and stability. *Journal of the Operational Research Society*, 49(3):237–252, March 1998.
[52] A. Koubaa and Y.-Q. Song. Evaluation and improvement of response time bounds for real-time applications under non-preemptive fixed priority scheduling. *International Journal of Production and Research*, 42(14):2899–2913, July 2004.
[53] P. Kumar. Re-entrant lines. *Queueing Systems*, 13:87–110, 1993.
[54] P. Kumar and T. Seidman. Dynamic instabilities and stabilization methods in distributed real-time scheduling of manufacturing systems. *IEEE Transactions on Automatic Control*, 35(3):289–298, March 1990.
[55] I. Kuroda and T. Nishitani. Asynchronous multirate system design for programmable dsps. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 5, pages 549–552, March 1992.
[56] J.-W. Lee, M. Chiang, and R. Calderbank. Jointly optimal congestion and contention control based on network utility maximization. *IEEE Communication Letters*, 10(3):216–218, March 2006.
[57] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 166–171, December 1989.
[58] J. Liebeherr, D. Wrege, and D. Ferrari. Exact admission control for networks with a bounded delay service. *IEEE/ACM Transactions on Networking*, 4(6):885 – 901, December 1996.
[59] X. Lin, N. Shroff, and R. Srikant. A tutorial on cross-layer optimization in wireless networks. *IEEE Journal on Selected Areas in Communications (JSAC)*, 24(8):1452–1463, August 2006.
[60] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of ACM*, 20(1):46–61, 1973.
[61] D. H. Lorenz and A. Orda. Optimal partition of qos requirements on unicast paths and multicast trees. *IEEE/ACM Transactions on Networking*, 10(1):102–114, February 2002.
[62] C. Lu, J. A. Stankovic, G. Tao, and S. Son. Design and evaluation of a feedback control edf scheduling algorithm. In *IEEE Real-Time Systems Symposium*, pages 56–67, 1999.
[63] C. Lu, X. Wang, and X. Koutsoukos. End-to-end utilization control in distributed real-time systems. In *ICDCS*, pages 456–466, 2004.
[64] S. Lu and P. Kumar. Distributed scheduling based on due dates and buffer priorities. *IEEE Transactions on Automatic Control*, 36(12):1406–1416, December 1991.
[65] J. Mo and J. Walrand. Fair end-to-end window-based congestion control. *IEEE/ACM Transactions on Networking*, 8(5):556–567, October 2000.
[66] M. J. Neely and E. Modiano. Capacity and delay tradeoffs for ad-hoc mobile networks. *IEEE Transactions on Information Theory*, 51(6):1917–1937, June 2005.
[67] Network Simulator, NS-2. `http://www.isi.edu/nsnam/ns/index.html`.
[68] J. Palencia and M. Harbour. Offset-based response time analysis of distributed systems scheduled under edf. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 3–12, July 2003.
[69] J. Palencia and M. G. Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *IEEE Real-Time Systems Symposium*, pages 26–37, December 1998.
[70] D. Palomar and M. Chiang. A tutorial on decomposition methods for network utility maximization. *IEEE JSAC*, 24(8):1439–1451, August 2006.
[71] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: The single-node case. *IEEE/ACM Transactions on Networking*, 1(3):344 – 357, 1993.
[72] T. M. Parks and E. A. Lee. Non-preemptive real-time scheduling of dataflow systems. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 5, pages 3235–3238, May 1995.
[73] R. Pellizzoni and G. Lipari. Improved schedulability analysis of real-time transactions with earliest deadline scheduling. In *IEEE Real-Time Applications Symposium (RTAS)*, pages 66–75, March 2005.
[74] S. Perathoner, N. Stoimenov, and L. Thiele. Reliable mode changes in real-time systems with fixed priority or edf scheduling. Technical Report TIK Report No. 292, ETH Zurich, Switzerland, September 2008.
[75] J. Perkins and P. Kumar. Stable, distributed, real-time scheduling of flexible manufacturing/assembly/disassembly systems. *IEEE Transaction on Automatic Control*, 34(2):139–148, February 1989.
[76] R. Racu, M. Jersak, and R. Ernst. Applying sensitivity analysis in real-time distributed systems. In *IEEE RTAS*, pages 160–169, 7-10 2005.
[77] K. Ramamritham. Allocation and scheduling of complex periodic tasks. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 108–115, 1990.
[78] J. Real and A. Crespo. Offsets for scheduling mode changes. In *ECRTS*, pages 3–10, June 2001.
[79] M. Saad, A. Leon-Garcia, and W. Yu. Rate allocation under network end-to-end quality-of-service requirements. In *IEEE Globecom*, pages 1–6, November 2006.
[80] J. Schneider. Cache and pipeline sensitive fixed priority scheduling for preemptive real-time systems. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 195–204, November 2000.
[81] L. Sha, R. Rajkumar, J. Lehoczky, and K. Ramamritham. Mode change protocols for priority-driven preemptive scheduling. *Real-Time Systems*, 1(3):243–264, December 1989.

[82] C. Shen, K. Ramamritham, and J. A. Stankovic. Resource reclaiming in multiprocessor real-time systems. *IEEE Transactions on Parallel and Distributed Systems*, 4(4):382 –397, April 1993.

[83] M. Spuri. Holistic analysis of deadline scheduled real-time distributed systems. Technical Report RR-2873, INRIA, France, 1996.

[84] J. A. Stankovic, C. Lu, S. Son, and G. Tao. The case for feedback control real-time scheduling. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 11–20, 1999.

[85] L. Tassiulas. Linear complexity algorithms for maximum througput in radio networks and input queued switches. In *INFOCOM*, pages 533–539, 1998.

[86] L. Tassiulas and A. Ephremides. Stability properties of constrained queueing systems and scheduling policies for maximum throughput in multihop radio networks. *IEEE Transactions on Automatic Control*, 37(12):1936 – 1948, December 1992.

[87] L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In *IEEE International Symposium on Circuits and Systems*, volume 4, pages 101–104, May 2000.

[88] K. Tindell, A. Burns, and A. Wellings. Mode changes in priority preemptively scheduled systems. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 100–109, December 1992.

[89] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Elsevier Microprocessing and Microprogramming*, 40(2-3):117–134, 1994.

[90] S. Verma, R. Pankaj, and A. Leon-Garcia. Call admission and resource reservation for guaranteed quality of service (gqos) services in internet. *Elsevier Computer Communications*, 21(4):362–374, April 1998.

[91] E. Wandeler, A. Maxiaguine, and L. Thiele. Quantitative characterization of event streams in analysis of hard real-time applications. In *IEEE Real-Time Applications Symposium (RTAS)*, pages 450–459, May 2004.

[92] A. Warrier, S. Janakiraman, S. Ha, and I. Rhee. Diffq: Practical differential backlog congestion control for wireless networks. In *INFOCOM*, 2009.

[93] J. Xu. Multiprocessor scheduling of processes with release times, deadlines, precedence, and exclusion relations. *IEEE Transactions on Software Engineering*, 19(2):139–154, February 1993.

[94] J. Xu and D. Parnas. On satisfying timing constraints in hard real-time systems. *IEEE Transactions on Software Engineering*, 19(1):70–84, January 1993.

[95] X. Yuan and A. K. Agrawala. A decomposition approach to non-preemptive scheduling in hard real-time systems. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 240–248, December 1989.

[96] N. Zhang, A. Burns, and M. Nicholson. Pipelined processors and worst case execution times. *Real-Time Systems*, 5(4):319–343, 1993.

[97] Y. Zhang, C. Lu, C. Gill, P. Lardieri, and G. Thaker. End-to-end scheduling strategies for aperiodic tasks in middleware. Technical Report WUCSE-2005-57, University of Washington at St. Louis, December 2005.

[98] G. Zhou, T. He, J. A. Stankovic, and T. Abdelzaher. Rid: Radio interference detection in wireless sensor networks. In *IEEE Infocom*, volume 2, pages 891–901, March 2005.